A Semantic Framework for Higher-Order Functional Logic Programming with Lambda Abstractions

TRABAJO FIN DE MÁSTER
en Programación y Tecnología Software

Fernando Pérez Morente

Directores:
Francisco Javier López Fraguas
Rafael del Vado Vírseda (colaborador externo)

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Curso 2010-2011[1]

[1]Trabajo presentado en la convocatoria de Junio de 2011 obteniendo la calificación de 10.

A Semantic Framework for Higher-Order Functional Logic Programming with Lambda-Abstractions

Memoria correspondiente al
Trabajo de Fin de Máster presentada por

Fernando Pérez Morente

Dirigida por los profesores

Francisco Javier López Fraguas y Rafael del Vado Vírseda

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Madrid, Junio de 2011

Autorización de Difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: **A Semantic Framework for Higher-Order Functional Logic Programming with Lambda Abstractions**, realizado durante el curso académico 2010-2011 bajo la dirección de Dr. Francisco Javier López Fraguas y Dr. Rafael del Vado Vírseda en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fernando Pérez Morente

20 de Junio de 2011

Abstract

Declarative programming is a programming paradigm with solid mathematical foundations that allow to design programs with a very high level of abstraction; the functional logic formalism puts together functional and logic formalisms and has been an intense matter of research in the last two decades. In this work we study a modern semantic framework for higher-order functional logic programming with λ-abstractions, as an extension to pattern rewriting systems based on λ-calculus to add higher-order features based on λ-abstractions and higher-order unification to standard functional logic programming languages. We present a declarative rewriting logic *GHRC* with an associated calculus with the same name that formally specifies derivability from the logic. Then we present declarative semantic concepts in the form of classic model-theoretic semantics and fixed-point semantics. Finally we present an extension of the framework to support modular construction of higher-order programs and we define semantics suitable for this extension proving that they are compositional and fully abstract with respect to the classical operations defined over modules.

Keywords

Declarative programming, Functional logic programming, Term rewriting systems, Pattern rewriting systems, Lambda calculus, Semantics of programming languages, Modular semantics.

Resumen en Castellano

La programación declarativa es un paradigma de programación con sólidos fundamentos matemáticos que permite diseñar programas con un alto nivel de abstracción; la programación lógico funcional aúna los formalismos lógico y funcional y ha sido un importante campo de investigación en las últimas décadas. En este trabajo se estudia un marco moderno para programación lógico funcional de orden superior con λ-abstracciones, que es una extensión de los sistemas de reescritura de patrones basada en λ-cálculo para añadir características de orden superior basadas en λ-abstracciones y unificación de orden superior a los lenguages de programación lógico funcional convencionales. En este trabajo presentamos la lógica de reescritura *GHRC* que tiene un cálculo asociado con el mismo nombre y que especifica formalmente la derivabilidad a partir de la lógica. Después presentamos conceptos semánticos en base a una semántica de modelos y una semántica de punto fijo. Finalmente, presentamos una extensión del marco para permitir la construcción modular de programas de orden superior y definimos una semántica modular adecuada para ello demostrando que es composicional y completamente abstracta con respecto a las operaciones definidas sobre módulos.

Palabras Clave

Programación declarativa, Programación lógico funcional, Sistemas de reescritura de términos, Sistemas de reescritura de patrones, Lambda cálculo, Semántica de los lenguages de programación, Semánticas modulares.

Notes about this work

This work has been done as a master's thesis during the course 2010-2011 in the context of the master's program of the 'Universidad Complutense de Madrid' in the speciality of 'Programación y Tecnología Software' corresponding to the 'Departamento de Sistemas Informáticos y Computación'.

This work has been supervised by Francisco Javier López Fraguas and directed by Rafael del Vado Vírseda in the context of the 'Declarative Programming Group' lead by the former.

Finally, this work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

Acknowledgements

I would like to acknowledge the following people because this work would not have been possible without their help:

- My director Paco López Fraguas for giving me the opportunity to work in his research group with Rafael and his constant support and care to all of my research activities. Also I would like to acknowledge him for his amazing courses that were an important reason for me to decide to try a research career.
- My director Rafa del Vado Vírseda for letting me work with him in his research activities. I have a lot of things to acknowledge Rafa for in this work (and even more outside of this), but I would like to specially mention his disposition to explain me and discuss, sometimes endless times, every concept and idea in this work, and his relentless efforts to review uncountable versions of each part of this work.
- All the members of the 'Departamento de Sistemas Informáticos y Computación' of the Universidad Complutense de Madrid', specially those from the 'Declarative Programming Group', and all my professors from the master's program for their exciting lectures, explanations and help provided for doing this work. Many of them are a constant inspiration to me.
- My colleagues from 'Aula 15' (yes, it is an awkward name for a place to work) for the great space to work they contribute to create.
- My parents and Sara for their generous and constant support in every aspect of my life.

Contents

Chapter 1

Introduction

In this chapter we define the scope of this Master's Thesis. First, we give a brief description of the *declarative programming* paradigm, explaining the basic concepts of the two most representative formalisms of the paradigm: *functional programming* and *logic programming*. The interest for combining both formalisms led to the emergence of the *functional logic programming* paradigm; even though the functional logic formalism have not succeeded in replacing the other two, it has been proved to be at the basis of useful languages with some interesting applications in prototyping, embedded systems, e-learning and web-based information systems.

This work covers a higher-order extension to conventional functional logic programming in order to integrate λ-*abstractions* and *higher-order unification* to it. This preliminary chapter gives a historical perspective of the works and efforts in these research areas, and describes some of the systems that have been implemented following these theoretical ideas.

We also provide a description of the field of *declarative modular programming*, that enriches declarative programming paradigms with modular constructions in order to support the modular design of declarative programs, that is an indispensable requirement for any programming language to manage the design of complex software and specific purpose libraries. This description serves as motivation to Section 4.3, that presents a modular extension with respect to the classical approach of our framework.

In the last section of this chapter we present a detailed description of the contents of this work, their contributions, and the main objectives that have been pursued in its development.

1.1 Higher-Order Functional Logic Programming

In this section we start describing the basis of the functional logic programming paradigm as an extension of functional programming and logic programming, which are the two most relevant paradigms in the field of declarative programming. There is a vast literature about that paradigms and here we only sketch their main characteristics necessary to understand the scope of this work. Then we present extensions developed for the combined paradigm to complete the framework with higher-order features.

1.1.1 Declarative Programming

Declarative programs are sets of statements implicitly defining a set of computations that can be performed but do not contain control information about how those computations should be performed; because of that they usually do not have statements to control the flow of instructions or direct management of a memory model. The task of the programmer in these languages is to define the logic of the program, leaving to the computational model the operational details about how the program is being computed. They are close to specification languages with the difference that they can be executed; that makes declarative programming good for prototyping and testing alternative solutions to problems with a very high level of abstraction.

Declarative programming languages have a higher level of abstraction than their imperative counterparts, so they are more expressive and unloads from the programmer a lot of work; on the other hand, apparently it is possible to write more efficient programs in an imperative programming language because of the capacity of directly controlling data memory accesses and because of the bigger complexity of the computational model; this is not necessary the case, and even though imperative programs tend to be more efficient than their declarative equivalents, advances in compiler design help to make these very expressive declarative programming languages more competitive in the "real world".

Declarative programs are sets of statements. In order to perform computations, those statements are interpreted with respect to a logic in a way such that computations are actually inferences in the logic. It is important to notice that even though computations can be inferred from the logic defining the system, proceeding this way is usually inherently inefficient; because of that, operational semantics propose efficient ways of performing those computations that can be inferred from the logic and it is an essential task to prove the equivalence between what can be inferred and what can be computed. Obviously, advanced compiler design techniques can be applied to the resulting program in order to improve efficiency even more, at the same level that is done to imperative languages; operational semantics try to minimize as

much as possible the number of logical steps needed in order to compute a goal, for example, avoiding computations that are not going to compute useful information or trying to select the most promising path to solve the problem; compiler designers deal with particular aspects of the machine where the program is going to be executed and can perform successive optimizations at different levels of abstraction.

Declarative programming paradigms are characterized by the logical inference mechanism they present, being the two most prominent functional programming based on rewriting logics and logic programming based on predicate logic applied to a subset of it such as Horn clauses.

1.1.2 Functional Programming

In the functional programming paradigm [53, 84] a program is a set of functions that can be applied to simple and complex data elements defined by equations that use recursion and case distinction. Functions in functional programming translate to the computation field the concept of *mathematical function*. Mathematical functions are mappings from the elements of a domain set to the elements of a range set; that is, functions relate each element in the domain set with an unique element of the range set. In functional programming languages, functions have a *type* that denotes both the domain and the range set, and a set of equations that acts as definition of the function that, making use of the resources of the language, effectively compose a program that computes the element of the range set that corresponds to each element in the domain set.

As explained before, a functional program is a set of functions defined by means of equations; in order to define both the domain and range sets effectively, functional languages provide the capacity of defining *data types*; a data type consists on a possibly infinite set of values that can act as the domain and the range of functions. Data types are usually built from *data constructors* that can have any number of arguments; for example, natural numbers can be represented in the *Peano style* using a data constructor '$zero$' with no arguments to represent number 0 and data constructor s with one argument to construct the other natural numbers (1 is represented as '$s(zero)$', 2 as '$s(s(zero))$') etc.

Functional programming languages perform computations by means of the evaluation of expressions, that consists on evaluating the range element corresponding to an element of the domain of the function (functions can obviously have any number of them and they are called the arguments of the function) that can be given by means of the result of the evaluation of another expressions and so on. The evaluation of an expression can lead to a non-termination situation, and this makes useful to consider functions in a functional language as partial functions that may

not be defined for each element of their domain. The way to evaluate expressions in functional languages is by means of a rewriting (also called reduction) process [84]. A rewriting process transforms a given expression into another equivalent one by applying a rule defining a function.

An interesting feature of functional programming languages is the way how arguments of functions are computed in the reduction process, that corresponds with the idea of strict function. An argument of a function is *strict* if its evaluation is necessary in order to denote the element on the range set corresponding to a set of arguments in the domain set and non-strict or *lazy* otherwise; for example, a function with two arguments that relates every pair of numbers with the first of them, is strict in the first argument because without knowing it we couldn't know the result of the evaluation but is non-strict on the other argument because its value is not relevant to know that result. An *eager* functional programming language will evaluate all the arguments of a function before computing a result while a lazy one will only evaluate those arguments that are really needed to proceed with the computation. Even though lazy evaluation seems to be more efficient because implies less computations, it also implies an overload of computations to be lazy and requires the use of efficient internal data structures in order to compete in efficiency with the eager one; in general, eager languages are more efficient than their lazy counterparts.

Another interesting characteristic of functional languages is that they are *higher-order*. That means that functions can have other functions as elements of their domain and range sets. This makes functional programming languages very expressive and together with function composition makes easy to express complex computations with very simple and compact expressions.

Denotational semantics of functional languages usually translate programs to algebras [53] being the denotation of a fixed program an initial algebra that satisfy the program in some way.

1.1.3 Logic Programming

In the logic programming paradigm [4, 57] programs are built from statements of a *logic* that is adequate to define efficient operational semantics in order to be adequate to be at the basis of a programming language. Usually is a subset of the *predicate logic*, and the most commonly used for logic programming are Horn clauses. If we consider that logic, a program is a set of Horn clauses that define relations, also called *predicates*, among the elements of a syntactic domain called the *Herbrand universe* [4] whose elements are called *terms*. Logic programs can contain *variables*, that represent generic elements from the domain and play an essential role in the formalism.

Logic programming languages perform computations by means of the evaluation of goals; a goal is a question about relations among elements in the domain, that are answered by checking if they can be deduced from the program and the inference rules from the logic. In this context, the answer to a goal not only consists on checking if the goal can be deduced from the program in the logic, but also contains the specific assignment of terms to the unknown information in the goal, in the form of variables. If there is more than one assignment to variables for a goal or it can be deduced from the logic in several ways, then several different answers are computed. Thus, logic languages are said to be *non-deterministic*, because they explore the whole space of possible answers computing all of them.

The operational semantics of logic languages based on Horn clauses is *SLD-resolution*, that is a methodology of proof by refutation that is sound and complete with respect to the logic underlying Horn clauses and is adequate for a programming language because is easily adapted to efficient implementations. This method uses a particular strategy to search for goals that can lead to non-termination [57].

Logic programs are usually non-typed and allow different modes of use. This is a good feature for code reuse, because a single relation can be used for different tasks depending on the arguments that contain unknown information when presented in a goal.

Denotational semantics of logic languages usually translate programs to models being the denotation of a fixed program the least model (called *least Herbrand model* in this context) that satisfy the program in some way [4].

1.1.4 Functional Logic Programming

The functional logic programming paradigm represents a conservative combination of both the functional and logic programming paradigms, in the sense that programs written in only one of them are expected to have the same behavior in the combined paradigm. Modern surveys about functional logic programming can be found in [3, 46, 87].

From the functional paradigm point of view, functional logic programming extends it by adding logical variables and non-determinism to it and, as a consequence, the possibility of making computations with unknown information and different modes of use of functions that can be used for easy code reuse; the addition of non-determinism and logic variables to the paradigm suppose a big difference with it, since a function applied to a fixed set of elements from its domain no longer represents necessarily one or less elements from the range set. From the logical paradigm

point of view, functional logic programming extends it by incorporating *type systems*, *lazy evaluation strategies* that support the management of infinite structures, and *higher-order* features that allow to manipulate functions as data.

In functional logic programming, programs operate on data by means of functions that manipulate data. A program is a set of equations defining the behavior of defined functions, usually called *rules* because they are applied only in one direction, from left to right. This corresponds closely to the way that functions are defined in the functional programming paradigm. The difference comes when we add the last ingredient to these definitions: variables. Variables can appear both in rules and in expressions to evaluate and they can be instantiated to different values in the computation process, each of them possibly leading to several different solutions in a similar fashion that it is done in the logic programming paradigm.

There are two main proposals for operational semantics for functional logic programming languages. One is *narrowing* [2, 22], a technique originally introduced in automatic theorem proving and first proposed for programming in [85] that is based in *rewriting with unification*, that is similar to rewriting in the context of functional languages but substituting pattern matching with unification so bindings for variables are generated in the process. The other one is *residuation* [45], that is similar to *SLD*-resolution suspending parts of the goal until enough information to perform a deterministic computation is generated. Considering these two main proposals, narrowing is usually preferred for the operational semantics of functional logic languages since residuation is a sound but non-complete method [45].

Denotational semantics of functional logic languages usually translate programs to models being the denotation of a fixed program the least model that satisfy the program in some precisely defined way [38].

There has been developed several functional logic programming languages and systems in the last decades, being the most relevant ones:

- $\mathcal{TOY}$. A language based on narrowing with strong theoretical foundations and advanced constraint management [62].
- *Curry*. A language based on narrowing combined with residuation for concurrent computations with syntax close to the functional language *Haskell*. It has similar applications to $\mathcal{TOY}$ and several application oriented libraries [44].
- *Mercury*. A functional logic language which is optimized for execution with a computational model where functions and predicates have distinct modes [92].
- *Oz*. A residuation based language that has a computational model that extends

the concurrent constraint programming paradigm with features for distributed programming and stateful computations [91].

1.1.5 Higher-Order Functional Logic Programming

The combination of logical variables and unification with the higher-order features from functional programming led to interest in the capability of performing unification of terms based not only in their syntax (which corresponds to the idea of *intensional equality* of functions) but in the semantics (that corresponds to the idea of *extensional equality*).

Functional programming languages are of higher-order by nature, that is, functions are allowed to manipulate other functions. Most functional programming languages also allow λ-abstractions in their syntax, with a syntax borrowed from the λ-*calculus* [43, 52] that stands for anonymous functions that are not needed to be declared and abstract a variable from a functional expression. λ-abstractions suppose a straightforward way of having a syntactic representation of a huge space of functions without declaring specifically them and with a very compact and simple syntax. Functional languages operate by pattern matching and λ-abstractions cause no problem to the reduction model.

Historically, there have been proposed several higher-order extensions to the logic paradigm. In [93] was supported the idea that logic languages are expressive enough to support higher-order features without extensions via an special *apply symbol* and *flattening*, but recognizing the lack of expressiveness of such an approach at the same time. Even though, many people argued that adding λ-abstractions and higher-order unification to the paradigm would lead to a more expressive and adequate approach for many tasks [77]. There were attempts to add λ-terms to logic languages to support higher-order features; the problem was that higher-order unification with λ-*terms* is undecidable in the general case. In 1991 there was proposed a subset of λ-terms where higher-order unification is decidable [70]. The λ-*prolog* language [71] incorporated this new higher-order patterns, has been oriented to constraints in [56] and has developed a very efficient implementation of higher-order logic programming in [82]. An extension to add functional notation to the *language Ciao-Prolog*, which is translated by a pre-processor to the logic language Prolog, has been proposed in [17].

In the functional and logic paradigms, the presence of logic variables and unification makes interesting to extend the language with higher-order unification. Due to its expressiveness [47], higher-order declarative programming with constraints is used for specification and verification of software, hardware circuits synthesis, machine learning, theorem proving systems, and symbolic computation with complex

algebraic structures in mathematics (for instance, the kernel of *Mathematica*^{TM} is a particular application of higher-order rewriting techniques with numeric constraints), providing the necessary level of abstraction for concise and natural formulations.

In an extension to a framework similar to ours, [39] proposed a higher-order extension without adding λ-abstractions to the language; even though the language is not as expressive as ours due to this lack, applicative expressions without λ-abstractions are expressive enough for many purposes. This approach has been extended in [61, 60] and is currently implemented in the $\mathcal{TOY}$ system.

With respect to frameworks with λ-abstractions, in [47, 81] is developed a framework with strong connections to ours developing a higher-order lazy narrowing calculus oriented to solve higher-order equations. In [42] is developed a higher-order functional logic language with λ-abstraction that has been used for distributed constraint solving [65]. Finally, our framework is an extension to the first-order framework presented in [39] and some features of it have appeared in [23, 24, 25].

1.2 Modular Declarative Programming

In this section we give a perspective of the extensions to declarative programming languages that has been done since the 1980s in order to add modularity to them. This section motivates Section 4.3, where we make our own proposal to add modularity to our language.

Modularity is a central issue in all programming paradigms motivated by the need of mastering the complexity inherent in the construction of complex programs and systems. Building complex software systems by composing code and combining existing components is a standard methodology of software development with an incremental knowledge organization, in which programs should be developed incrementally by defining several units with their corresponding interfaces and then by composing those units. This leads to an approach to modularity based on the notion of program composition, where the effectiveness depends on the possibility of reasoning on the composition process itself. The availability of well-founded modular semantics of programs and program composition is important to perform sound *semantics-based transformation* [18], *analysis based on abstract interpretation* [58], *debugging* [55], and *verification* [80].

In the multi-paradigm declarative programming setting (see e.g., [46]), the combination of features like *higher-order*, *polymorphism* or *constraints* make essential to separate the whole task of designing a program into subtasks of manageable size.

Complex declarative programs are constructed in a structured way by combining and modifying smaller programs or components.

The interest in well-founded modular semantics for declarative programming languages and systems has motivated a considerable research effort over the past decades and it has been the subject of an active and still open research in the theory and practice of declarative programming. A number of different approaches to modularizing declarative programming systems and languages have received theoretical treatment and have been included in practical systems [41, 48, 50, 88]. Some of them augment declarative programming with constructs for declaring and using modules, much in the spirit of conventional programming languages. In this field, a typical module consists of a body, an export interface, a list of imports and, possibly, a list of formal parameters, and typical operations with modules have to do with setting up hierarchical relationships between modules as the union of modules or deletion of signatures and the application of a parameterized module to an actual module.

In the logic programming field, several compositional semantics for logic programs have been studied by using higher-order semantics and notions such as *full abstraction* [33, 36, 64]. Moreover, modularity has been the objective of different proposals which basically have followed two different guidelines. One, focused on *programming-in-the-large*, extends logic programming with modular constructs as a meta-linguistic mechanism [13] and gives semantics to modules with the aid of the immediate consequence operator. And the other one, focused on *programming-in-the-small*, enriches the theory with new logical connectives for dealing with modules [69]. In the functional logic programming paradigm, [73, 74] develops a study of program structuring and modularity in the first-order rewriting logic *CRWL*, based on a meta-linguistic mechanism close to that developed in [12, 13]. However, in the higher-order functional logic programming framework with constraints adopted by real multi-paradigm systems such as $\mathcal{TOY}$, we do not know any study of modularity semantically well-founded. The lack of modularity in this framework derives from the lack of compositionality in its standard semantics (i.e., the meaning of a program can be obtained from the meaning of its components).

1.3 Objectives and Organization

The main goal of this work is the study of a new semantic framework for higher-order functional logic programming with λ-abstractions and higher-order unification. There has been a lot of research work developed in this field in the last five years in the context of our research group that led to the development of the framework that is introduced in this work in order to develop several applications on the fields of

formal verification, algorithmic debugging, program transformation and the implementation of suitable operational semantics. This research leds to the publication of several scientific works [23, 24, 25, 26], that contain some kind of brief descriptions of the framework but are focused in the specific application they propose. In this work we focus in the declarative semantic framework itself, so it can complement all the other works by providing an extensive description of the framework that is behind or underlying most of them.

In this work we have tried to give an extensive and comprehensive description of the framework, with several examples and references to related work. We also compile essential semantic results that are needed to relate different notions on semantics that are proposed. Also, we have worked to complete the framework in some aspects that were not published before in the form of a new higher-order unification algorithm on patterns proposed in Appendix A and the modular semantics from Chapter 4.

This work is organized in five chapters and an appendix. The first chapter is an introductory chapter and the last one is a compilation of the main results of this work together with conclusions and future research topics that can be followed from it. The material relegated to the appendix is not essential to follow the rest of the work; even though, we think that this material is relevant in the field and have been included for the sake of completeness of the work, but in order to not disturb the flow of the work we have relegated it to an appendix.

- Chapter 1 contains an introduction to the field of functional logic programming and the most relevant efforts to add λ-abstractions and higher-order unification to those kind of languages; there is also a more specific discussion of modular programming that motivates Section 4.3 that contains a modular extension of the framework. Also we describe the most relevant objectives and the organization of the rest of the work.

- Chapter 2 consists on an extensive introduction to term rewriting systems and simply-typed λ-calculus as the basis of our framework, that is presented afterwards. In this initial chapter we introduce many of the concepts with their respective notations that are used intensively in the following sections and chapters. The introduction we give of those classic frameworks is oriented by what we need for the rest of the work, and we almost ignore aspects that are very relevant for each of them because they do not adjust to our framework, and pay special attention to concepts that are more tangential in those contexts but are of special interest in ours. After that we proceed to define the functional logic programming language with λ-abstractions that we are studying adapting those concepts defined before for term rewriting systems

and λ-calculus, making explicit the common notations that we use through the rest of the work and defining new concepts that are relevant only in the context of our framework.

- In Chapter 3 we present the kind of computations that can be performed from our language in the form of reduction or equality statements that can be proved from programs of the programming language described in Chapter 2 by means of a suitable rewriting logic with an associated proof calculus. These ideas are an extension to those for a first-order framework given in [38] and originally presented in [23, 24]. The proof calculus is not intended to be an operational model but denotes all the computations that can be performed from our language fixing formally its behavior. We also present a compilation of some basic results about the proof calculus that are essential in order to prove the central semantic theorems presented in Chapter 4.

- In Chapter 4 we provide the declarative foundations of our framework in the form of a *model-theoretic semantics*, that denotes the meaning of our programs as mathematical objects in the classical *domain theory*, and a *fixed-point semantics*, that provides a view of the model-theoretic semantics as the least fixpoint of a continuous operator defined over pattern algebras. We present essential equivalence results about those semantics that are usual in the context of declarative programming. Finally we present a modular extension of our programming language by defining a suitable modular semantics that is *compositional* and *fully abstract* [13, 60] with respect to the pattern model of the program as the corresponding notion of *observable*.

Finally, in Appendix A we describe a higher-order unification algorithm and we prove its adequateness for this framework. This appendix contains material that is more technical than the rest of this work.

The modular extension to this framework adapted in Section 4.3 applied to generic constraint domains has been accepted for presentation and publication [27] in the 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2011) that will take place on July 20-22 of 2011, in Odense, Denmark.

Chapter 2

A Functional Logic Programming Language with λ-Abstractions

The aim of this chapter is to define the syntax and basic concepts about the higher-order functional logic programming language with λ-abstractions that is the object of study of this work. The first two sections of the chapter give some basic notions of two classical and well-known frameworks that are at the basis of our work: term rewriting systems and λ-calculus; the goal here is only to give the notations and the basic definitions that will be used in the rest of the work; we are not intending to study in depth these frameworks; only a brief description of each of them is given and no results are proved (for additional information about these frameworks we submit the readers to the bibliographical resources that are referenced in its corresponding section). In the last section of the chapter we describe in detail the syntax and some basic concepts about the language we are studying. We will adapt and complete some of the definitions given in the previous sections, that will be useful when defining logics and semantics for the programming framework in the following chapters.

2.1 Term Rewriting Systems

In this section we present some basic concepts of Term Rewriting Systems. Abstract rewriting systems are widely adopted in Mathematics to describe systems where entities are changing by means of some defined rules, such as groups, ideals and fields [10]. In the field of Computation, a widely adopted view of a computation process is the one that models it as a sequence of state transitions; this view can be translated to a framework where states are represented as *expressions* and transitions are represented as *rewrite rules* that can be applied to that expressions,

so they can be straightforwardly modeled as an abstract rewriting system that receive the name of *term rewriting system*. Term rewriting systems are a very simple but Turing-complete computational model, with very simple syntax and operational semantics; in spite of its simplicity, concepts and methods developed for them have been proved relevant in other Computer Science areas. Deep studies about term rewriting systems that cover in depth all the topics mentioned in this section and additional interesting ones can be found in [5, 8, 11, 54].

In the context we are working from now on, expressions of the rewriting system are called *terms*, that give a defined syntactic structure to expressions and provide a well-known model from universal algebra to the syntactic level of the rewriting systems. Transitions are modeled by means of rewriting rules, which have a *left-hand side* and a *right-hand side*; if a given term (or just a subterm of it) matches in some way, that will be precisely defined later, with the left-hand side of a rewriting rule, then the term (or a subterm) can be rewritten into a variation of the right-hand side of the rule. The process of sequentially applying rewriting rules to a term and the terms generated in the process gets the name of *reduction*, because usually the process keeps reducing the complexity of the term considered in some way; this is not always the case, but that is the reason why the name is given. The set of all the rewriting rules inducts a *reduction relation* which specifies all the valid reduction sequences in the system.

2.1.1 Syntax and Structure of Terms

To begin the study of term rewriting systems we are going to precisely define its most basic syntactic construction: *terms*. Terms are structured syntactic elements that are built from a set of symbols called the *signature* of the term rewriting system. Each symbol in the signature can be applied to a fixed number of other terms (that might be zero) in order to build new terms; this number of *input arguments* is called its *arity* and in our framework is considered to be unique; that is, we cannot have an overloaded symbol with two different arities. Symbols of the signature will also be referred as *function symbols*.

Definition 2.1.1 (Signature) *A* ***signature*** Σ *is a set of symbols where each* $f \in \Sigma$ *is associated to a non-negative integer* n *that is the* ***arity*** *of* f*, denoted as* $arity(f) = n$ *or* f/n*. For each* $k \geq 0$*, the set of all symbols of arity* k *is denoted as* Σ^k*. The elements of* Σ^0 *are called* ***constant symbols****.*

From the elements of the signature, terms are built by applying function symbols to other terms. Most basic terms are *constant* symbols, that correspond to function symbols with no arguments, and another basic structural component, the so-called *variables*. A variable is a symbol that has no arity associated because they are not

applied to other terms in order to form new ones. A variable has a property that differences it from a constant symbol: it can be *instantiated* by any other term in the rewriting process that is defined later.

Definition 2.1.2 (Terms) *Let Σ be a signature and $\mathcal{V}$ a set of symbols, called **variables**, such that $\Sigma \cap \mathcal{V} = \emptyset$. The set of all Σ-**terms** over $\mathcal{V}$, denoted as $\mathcal{T}(\Sigma, \mathcal{V})$, is defined inductively as follows:*

- $\forall v \in \mathcal{V}.v \in \mathcal{T}(\Sigma, \mathcal{V})$.
- $\forall f \in \Sigma^n.f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ *if* $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$.

From an operational point of view variables behave in a special way; so that, it is useful to define the set of all variables that appear in a term. It is straightforward to define this set by induction over the structure of the term. Terms that have no variables occurring in them are called *ground terms*.

Definition 2.1.3 (Variables in a term) *Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$. The **set of variables** of t is denoted as $Var(t)$ and defined inductively over the structure of t as follows:*

- *if* $t \in \mathcal{V}$, $Var(t) = \{t\}$.
- *if* $t \in \Sigma^0$, $Var(t) = \emptyset$.
- *if* $t = f(t_1, \ldots, t_n)$, $f \in \Sigma^n$ $(n > 0)$, *and* $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$, $Var(t) = \bigcup_{i \in \{1,\ldots,n\}} Var(t_i)$.

*Term t is a **ground term** if verifies that $Var(t) = \emptyset$.*

Example 2.1.4 (Terms) *Through this chapter we are going to study a term rewriting system for basic management of the Peano natural numbers, built from a constant symbol 'zero' and a successor symbol 's'. We define operations for addition ('add') and multiplication ('mult'). We also represent boolean numbers from two constant symbols ('true' and 'false') with a negation operation ('not'). Finally, we have a function to compare natural numbers ('leq'). A signature containing these symbols is defined in the following way:*

$$\Sigma = \{zero/0, true/0, false/0, s/1, not/1, add/2, mult/2, leq/2\}$$

In this signature, $\Sigma^0 = \{zero/0, true/0, false/0\}$, $\Sigma^1 = \{s/1, not/1\}$ and $\Sigma^2 = \{add/2, mult/2, leq/2\}$. Also holds that $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 = \Sigma$ and $\Sigma^0 \cap \Sigma^1 \cap \Sigma^2 = \emptyset$.

From the signature Σ and a set of variables $\mathcal{V}$ such that $\Sigma \cap \mathcal{V} = \emptyset$ we can build terms; to make sure that the signature and the set of variables are disjoint, is usually adopted the notation that elements from Σ starts with a lowercase letter and elements from $\mathcal{V}$ start with an uppercase letter; we do this in this section. For example, with $\mathcal{V} = \{X, Y, Z\}$, we can build some terms in $\mathcal{T}(\Sigma, \mathcal{V})$:

- $t_1 \equiv add(s(zero), s(s(zero)))$ *to represent the addition of* 1 *and* 2.
- $t_2 \equiv mult(s(X), Y)$ *to represent the multiplication of a natural number greater than* 0 *and a natural number greater or equal to* 0.
- $t_3 \equiv not(leq(X, Y))$ *to represent the negation of the fact that a natural number* X *is less or equal than a natural number* Y.
- $t_4 \equiv leq(true, s(zero))$ *is a term with no meaning in our intended interpretation of the symbols in the signature. Even though is a well-constructed term from a syntactic point of view, we do not have in mind the possibility of comparing a boolean value with a natural number.*
- $t_5 \equiv not(true, false, Z)$ *is not in* $\mathcal{T}(\Sigma, \mathcal{V})$. *Term* t_5 *is built from symbols of* Σ *and* $\mathcal{V}$, *but the symbol 'not' is applied to three arguments when its arity is one.*

Even though we write those symbols with an intended meaning in mind, at the syntactic level it is not possible to give it to them, and terms that in our pretended meaning would not be valid such as t_4 *are accepted from a syntactic point of view. But other terms with no meaning are rejected because of the arity condition checked when building terms, which is the case of* t_5. *When we define the rewrite rules of the term rewriting system we give the desired meaning to the terms defined. Obviously, there will not be any rule that can be applied to a term like* t_4, *but if we would like to explicitly reject it we need to attach a type system (see [51]) to the term rewriting system.*

Finally, we evaluate the set of variables of the terms defined before:

- $Var(t_1) = \emptyset$.
- $Var(t_2) = \{X, Y\}$.
- $Var(t_3) = \{X, Y\}$.
- $Var(t_4) = \emptyset$.

Both, t_1 *and* t_4, *are ground terms.*

A basic concept when reasoning about terms is the notion of *subterm*. Terms are defined in such a way that from some basic components (variables and constant symbols), more complex terms are constructed by applying function symbols as many times as desired. It is useful to have the capacity of exploring the way a term has been constructed, identifying its structure and those more basic components that put together generate the full term; we call these structural constructions subterms and the original term is one of them; the others are the terms that combined by the

application of function symbols generate it.

To be able to precisely identify the location of a subterm in a given term, it is necessary the concept of *position*. A position is a sequence of integer numbers that is used to traverse the original term in order to locate the exact subterm the sequence is referring to. The *empty sequence* is represented as ϵ and corresponds to the original term; sequences are represented as lists of integer numbers separated by the $\cdot$ symbol: $n_1 \cdot n_2 \cdot \ldots \cdot n_k$. Each integer in the sequence identifies the order of the argument in the term that is being considered, where is located the subterm that the sequence refers to. That is, if we consider a subterm of the form $f(t_1, \ldots, t_n)$ and we look for the subterm represented by a sequence $i \cdot s$, then we have to look for the subterm represented by the sequence s in the ith-argument of f, that is, the subterm t_i; if s happens to be the empty sequence ϵ, the subterm we are looking for is just the term t_i.

The first of the following definitions formalizes the set of valid positions in a given term, that is, all the sequences of integer numbers that correspond to a subterm; it also defines the set of positions whose associated subterm is not a variable and receives the name of *ground positions*; this concept will be useful later when defining the rewriting process. The second definition defines a mapping from all these valid positions into the subterm of the original term they respectively refer following the idea described before.

Definition 2.1.5 (Positions)

1. *The set of* ***positions*** *in a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted as $Pos(t)$ and is defined inductively as follows:*

   - $\epsilon \in Pos(t)$.
   - *if* $t = f(t_1, \ldots, t_n)$, $arity(f) = n$ $(n > 0)$, *and* $i \in \{1, \ldots, n\}$ *then* $\forall x \in \{i \cdot p \,|\, p \in Pos(t_i)\}.x \in Pos(t)$.

2. *The set of* ***ground positions*** *in a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted $GPos(t)$ and is defined inductively as follows:*

   - $\epsilon \in GPos(t)$ *if* $t \notin \mathcal{V}$.
   - *if* $t = f(t_1, \ldots, t_n)$, $arity(f) = n$ $(n > 0)$, *and* $i \in \{1, \ldots, n\}$ *then* $\forall x \in \{i \cdot p \,|\, p \in GPos(t_i)\}.x \in GPos(t)$.

Definition 2.1.6 (Subterm at a position) *If $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and $p \in Pos(t)$, the* ***subterm of*** *t* ***at position*** *p is denoted as $t|_p$ and is defined by induction over the position p:*

- $t|_\epsilon = t$.
- $f(t_1, \ldots, t_n)|_{i \cdot q} = t_i|_q$.

Sometimes we want to replace a subterm of some given term, identified by its position, by another term. An useful notation for this is defined as follows:

Definition 2.1.7 (Substitution of a subterm at a position)
*Let $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ and $p \in Pos(t)$. The **substitution of the subterm in position** p of t by s is denoted $t[s]_p$ and is defined by induction over the position p as follows:*

- $t[s]_\epsilon = s$.
- $f(t_1, \ldots, t_i, \ldots, t_n)[s]_{i \cdot p} = f(t_1, \ldots, t_i[s]_p, \ldots, t_n)$.

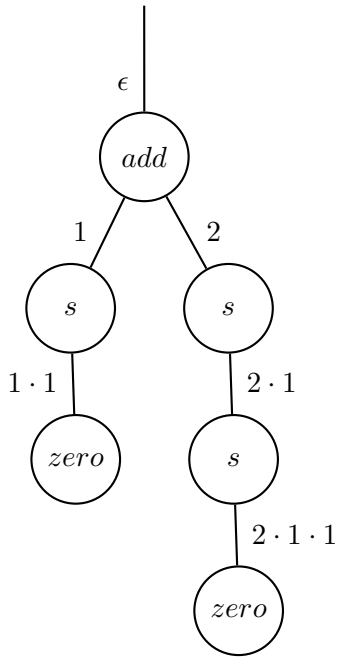
Example 2.1.8 (Subterms) *The set of positions for $t_1 \equiv add(s(zero), s(s(zero)))$ from Example 2.1.4 is $Pos(t_1) = \{\epsilon, 1, 2, 1 \cdot 1, 2 \cdot 1, 2 \cdot 1 \cdot 1\}$ and the subterm corresponding to each of them is:*

$p \in Pos(t_1)$	$t_1\|_p$
ϵ	$add(s(zero), s(s(zero)))$
1	$s(zero)$
2	$s(s(zero))$
$1 \cdot 1$	$zero$
$2 \cdot 1$	$s(zero)$
$2 \cdot 1 \cdot 1$	$zero$

The subterm corresponding to each position has been easily evaluated following Definition 2.1.6; for example, to evaluate $t_1|_{2 \cdot 1 \cdot 1}$ we have followed the following steps:

$$add(s(zero), s(s(zero)))|_{2 \cdot 1 \cdot 1} = s(s(zero))|_{1 \cdot 1} = s(zero)|_1 = zero|_\epsilon = zero$$

The evaluation of the subterm corresponding to a given position is similar to the process of traversing the tree corresponding to the term that has its subtrees numbered in the same way that positions are defined, that is a natural and common way of numbering trees:



The set of ground positions of t_1*,* $GPos(t_1)$*, verifies that* $GPos(t_1) = Pos(t_1)$ *because* t_1 *is a ground term. Always that a term is ground its set of positions and its set of ground positions are exactly the same since it contains no variables at all.*

For the term $t_2 \equiv mult(s(X), Y)$*, also from Example 2.1.4, the set of positions is* $Pos(t_2) = \{\epsilon, 1, 2, 1 \cdot 1\}$ *and the subterm corresponding to each of them is:*

$p \in Pos(t_2)$	$t_2\|_p$
ϵ	$mult(s(X), Y)$
1	$s(X)$
2	Y
$1 \cdot 1$	X

In the case of t_2*,* $GPos = \{\epsilon, 1\}$ *because position* $1 \cdot 1$ *corresponds to subterm* X *and position* 2 *corresponds to subterm* Y*, that are both variables. This example only verifies that* $GPos(t_2) \subseteq Pos(t_2)$*, that always holds for any term.*

2.1.2 Substitutions and Unification of Terms

A very useful concept when studying term rewriting systems is the concept of *substitution*. Substitutions are functions that map variables into terms, but only a finite

number of the elements in $\mathcal{V}$ are mapped into terms different of them; every other variable in the term rewriting system is mapped to itself by the substitution. Substitutions can be applied to any term in order to get another term, just by applying simultaneously the substitution to all the variables that appear in the term.

Definition 2.1.9 (Substitution) *Let $\mathcal{T}(\Sigma, \mathcal{V})$ be a set of terms. A **substitution** is a function $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ that verifies $\sigma(v) \neq v$ for only a finite number of variables in $\mathcal{V}$; the set of these variables is called the **domain** of σ and is denoted as $Dom(\sigma)$. If $Dom(\sigma) = \{x_1, \ldots, x_n\}$ and $\forall i \in \{1, \ldots, n\}.\sigma(x_i) = t_i$ then σ is represented as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.*

The substitution that is the identity function over the variables of the term rewriting system is called the *empty substitution* and is denoted as ε. Substitutions can be compound as any function: if $\sigma, \tau : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ then the composition of σ and τ is denoted by $\sigma\tau$ and is defined as the substitution that maps each variable $v \in \mathcal{V}$ to $\sigma(\tau(v))$. If there exists a substitution ρ such that $\tau = \sigma\rho$ then we say that σ is *more general* than τ and denote it as $\sigma \leq \tau$, defining an order relation over the set of substitutions, being the empty substitution the minimum element of this order. Sometimes will be useful to define a substitution σ from a more concrete one τ by restricting its domain to a set of variables $V \subseteq \mathcal{V}$ such that $\sigma(v) = \tau(v)$ if $v \in V$ and $\sigma(v) = v$ if $v \notin V$; we denote it as $\sigma = \tau \restriction_V$ and $\sigma \restriction_V \leq \tau \restriction_V$ as $\sigma \leq \tau\,[V]$.

Definition 2.1.10 (Application of a substitution to a term) *Let $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ be a substitution. The application of σ to t is denoted as $\sigma(t)$ and is defined by induction over the structure of t as follows:*

- *if $t \in \mathcal{V}$, $\sigma(t) = \sigma(t)$, considering the latter occurrence of σ as the value of the substitution for the variable t.*
- *if $t \in \Sigma^0$, $\sigma(t) = t$.*
- *if $t = f(t_1, \ldots, t_n)$, and $arity(f) = n$ $(n > 0)$, $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.*

Example 2.1.11 (Substitutions) *We define three substitutions for the term rewriting system of Example 2.1.4 where $\mathcal{V} = \{X, Y, Z\}$. Then we apply them to $t_1 \equiv add(s(zero), s(s(zero)))$, $t_2 \equiv mult(s(X), Y)$, and $t_3 \equiv not(leq(X, Y))$:*

$$\sigma_1 = \{X \mapsto Z\}$$
$$\sigma_2 = \{X \mapsto Z, Y \mapsto s(zero)\}$$
$$\sigma_3 = \{Y \mapsto zero, Z \mapsto s(X)\}$$

We can compose each pair of substitutions in order to obtain a new substitution:

$$\sigma_2\sigma_1 = \{X \mapsto Z, Y \mapsto s(zero)\}$$
$$\sigma_3\sigma_1 = \{X \mapsto s(X), Y \mapsto zero, Z \mapsto s(X)\}$$
$$\sigma_1\sigma_2 = \{X \mapsto Z, Y \mapsto s(zero)\}$$
$$\sigma_3\sigma_2 = \{X \mapsto s(X), Y \mapsto s(zero), Z \mapsto s(X)\}$$
$$\sigma_1\sigma_3 = \{X \mapsto Z, Y \mapsto zero, Z \mapsto s(Z)\}$$
$$\sigma_2\sigma_3 = \{X \mapsto Z, Y \mapsto zero, Z \mapsto s(Z)\}$$

This example illustrates the fact that the composition of substitutions is not commutative, analogously to the more general context of the composition of functions; in our example we have that $\sigma_3\sigma_1 \neq \sigma_1\sigma_3$. Also we have that σ_1 is more general than σ_2, that is $\sigma_1 \leq \sigma_2$, because $\sigma_2 = \{Y \mapsto s(zero)\}\sigma_1$. We cannot compare with that order relation σ_2 and σ_3 because $\sigma_2(Y) = s(zero)$ and $\sigma_3(Y) = zero$: we cannot find a substitution ρ such that $\rho(zero) = s(zero)$ or $\rho(s(zero)) = zero$. Thus the ordering between substitutions is a partial order and not every pair of substitutions can be compared pairwise by it.

Now we can apply the substitutions we have defined to t_1, t_2 and t_3; the results are in the following table, where each cell contains the result of applying the substitution in the row to the term in the column:

	$t_1 \equiv add(s(zero), s(s(zero)))$	$t_2 \equiv mult(s(X), Y)$	$t_3 \equiv not(leq(X, Y))$
$\sigma_1 = \{X \mapsto Z\}$	$add(s(zero), s(s(zero)))$	$mult(s(Z), Y)$	$not(leq(Z, Y))$
$\sigma_2 = \{X \mapsto Z, Y \mapsto s(zero)\}$	$add(s(zero), s(s(zero)))$	$mult(s(Z), s(zero))$	$not(leq(Z, s(zero)))$
$\sigma_3 = \{Y \mapsto zero, Z \mapsto s(X)\}$	$add(s(zero), s(s(zero)))$	$mult(s(X), zero)$	$not(leq(X, zero))$

As we can see, t_1 remains unchanged by applying any substitution to it and in general holds that any substitution maps every ground term to itself. This is obvious, since in a ground term there are no variables to replace.

Another useful concept in term rewriting systems is the one that defines equality among terms. Different equality notions can be defined and each of them is useful for different purposes. The most basic one is called *syntactic equality*: two terms s and t are equal by this notion if they are exactly the same sequence of symbols. The syntactic equality of two terms s and t is represented as $s \equiv t$. When defining how rewrite rules are applied to terms, we are interested in substitutions that make a term to be syntactically equal to the left-hand side of the rule. Any substitution that makes two terms s and t to be syntactically equal when applied to both of them is called an *unifier* of s and t.

Definition 2.1.12 (Unification) *Two terms s and t are **unifiable** if there exists a substitution σ such that $\sigma(s) \equiv \sigma(t)$. Then σ is called an **unifier** of s and t. If σ verifies $\sigma \leq \tau$ for each unifier τ of s and t then σ is the **most general unifier** (shortly, m.g.u.) of s and t.*

The process of finding the most general unifier of two given terms is called *unification*; this process is decidable and a detailed unification algorithm can be found in [5] (the first syntactic unification algorithm is in [49] and the algorithm referenced here follows the presentation of [66]).

Example 2.1.13 (Unifiers)

- *Substitutions* $\sigma_1 = \{X \mapsto zero, Z \mapsto zero\}$ *and* $\sigma_2 = \{X \mapsto zero, Y \mapsto Z, Z \mapsto zero\}$ *are unifiers of* $t_1 \equiv mult(X, s(zero))$ *and* $t_2 \equiv mult(zero, s(Z))$. $\sigma_1 \leq \sigma_2$ *because* $\sigma_2 = \{Y \mapsto Z\}\sigma_1$. *Moreover,* σ_1 *is the most general unifier of* t_1 *and* t_2*; intuitively this means that* σ_1 *contains the minimum information necessary to unify them.*

- *Substitutions* $\sigma_1 = \{Y \mapsto X, Z \mapsto zero\}$ *and* $\sigma_2 = \{X \mapsto zero, Y \mapsto zero, Z \mapsto zero\}$ *are unifiers of terms* $t_1 \equiv add(X, mult(Z, Z))$ *and* $t_2 \equiv add(Y, mult(zero, zero))$. *Moreover* $\sigma_1 \leq \sigma_2$ *because* $\sigma_2 = \{X \mapsto zero, Y \mapsto zero\}\sigma_1$. *In this example variables* X *and* Y *can be instantiated to a common term, and the more simplest substitution is the one that makes one of them to be the other.*

- *There not exists an unifier for* $t_1 \equiv mult(X, s(Y))$ *and* $t_2 \equiv mult(s(Y), zero)$. *This is because there is a constructor clash in the second argument of the function symbol 'mult', and there is no substitution that can make* $s(Y)$ *and* $zero$ *to be syntactically the same.*

Different equality notions or more general notions to compare terms will be employed through the rest of this work. Those notions will be useful for different purposes and all of them are conditions that can be fulfilled or not by each pair of terms. The most basic of these notions is *syntactic equality* and has been defined before; another common notion is the one that states that two terms are equal if and only if they can be rewritten to a common term by means of rewrite rules and is precisely defined in next section. It is useful to give a generic definition for all those notions so we can refer to them in a generic way. When defining the reduction process later on, the equality notion employed is the one that refers to reduction to a common term, but it could be different depending on extensions of the theory on term rewriting systems; for example, it could be referred to solving a set of constraints over a constraint domain (see [7, 35]). We will refer to this generic comparisons among terms as *equations* and we use the symbol $\approx$ for them. A calculus for solving equational problems can be found in [9, 22].

Definition 2.1.14 (Equation) *Let* $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. *The pair* (s, t) *is an* ***equation*** *and is represented as* $s \approx t$.

2.1.3 The Reduction Relation

After all these preliminaries, we formally define *rewrite rules* and the reduction relation induced by them. There are two kind of rewrite rules: *unconditional rules*, that are defined as pairs of terms and are applied straightforwardly in the reduction process, and *conditional rules*, that have a conditional argument represented by a set of equations that need to be fulfilled before applying the rule in the same way as unconditional ones.

Definition 2.1.15 (Rewrite rule) *Let $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$:*

1. *The pair (s, t) is a **rewrite rule** defined over $\mathcal{T}(\Sigma, \mathcal{V})$ if $s \notin \mathcal{V}$ and $Var(t) \subseteq Var(s)$, and is denoted as $s \to t$.*

2. *A tuple (s, t, C) is a **conditional rewrite rule** if (s, t) is a rewrite rule and C is a set of equations over terms in $\mathcal{T}(\Sigma, \mathcal{V})$, and is denoted as $s \to t \Leftarrow C$.*

We distinguish two kinds of term rewriting systems: those that only contain unconditional rules are called term rewriting systems; those that contain conditional rules (and maybe also unconditional ones) are called *conditional term rewriting systems*. An unconditional rule can be easily represented as a conditional one just leaving the conditional component of the conditional rewrite rule empty.

Definition 2.1.16 (Term rewriting system) *A **(conditional) term rewriting system** $\mathcal{R}$ is a pair (Σ, R) where Σ is a signature and R is a set of (conditional) rewrite rules over $\mathcal{T}(\Sigma, \mathcal{V})$.*

Now we explain how the *rewriting* process works by means of defining the *reduction relation* among terms. A pair of terms will be in the reduction relation if and only if the application of the rewrite rule to the first element of the pair, also called the *left-hand side* of the rule, leads to the second element or *right-hand side* of the rule. A rewrite rule $l \to r$ is applied to a term t following these steps: first of all, a subterm t' of t is matched with l by means of a substitution σ; then the substitution σ is applied to r obtaining a new term r'; finally, the subterm t' is substituted in t by r' in order to obtain the result of the application of the rewrite rule. Even though variables unify with the left-hand side of every rule, it is not allowed to apply rewrite rules to variable positions.

Definition 2.1.17 (Reduction relation)

1. *Let $\mathcal{R} = (\Sigma, R)$ be a term rewriting system and $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. The **reduction relation** $\to_{\mathcal{R}}$ is defined as follows: $s \to_{\mathcal{R}} t$ if and only if there exists a rewrite rule $(l \to r) \in R$, a position $p \in GPos(t)$, and a substitution σ such that $s|_p = \sigma(l)$ and $t = s[\sigma(r)]_p$. We call $s \to_{\mathcal{R}} t$ a **rewrite step**.*

2. *Let $\mathcal{R} = (\Sigma, R)$ be a conditional term rewriting system and $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. The* ***conditional reduction relation*** $\rightarrow_{\mathcal{R}}$ *is defined as follows: $s \rightarrow_{\mathcal{R}} t$ if and only if there exist a conditional rewrite rule $(l \rightarrow r \Leftarrow C) \in R$, a position $p \in GPos(s)$, and a substitution σ such that $s|_p = \sigma(l)$, $t = s[\sigma(r)]_p$, and every equation $(a \approx b) \in C$ is fulfilled in $\mathcal{R}$ (i.e., there exists $c \in \mathcal{T}(\Sigma, \mathcal{V})$ for each equation $a \approx b$ in C such that $\sigma(a) \rightarrow^*_{\mathcal{R}} c$ and $\sigma(b) \rightarrow^*_{\mathcal{R}} c$). We call $s \rightarrow_{\mathcal{R}} t$ a* ***conditional rewrite step****.*

The set of all possible reduction sequences in a (conditional) term rewriting system $\mathcal{R}$ is the reflexive transitive closure of the reduction relation $\rightarrow^*_{\mathcal{R}}$; the set of reductions sequences in one or more steps denoted as $\rightarrow^+_{\mathcal{R}}$. Any term that can not be rewritten by means of the reduction relation is called a *normal form*.

Definition 2.1.18 (Normal form) *Let $\mathcal{R} = (\Sigma, R)$ a (conditional) term rewriting system and $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. Term t is a* ***normal form*** *in $\mathcal{R}$ if and only if there not exists $r \in \mathcal{T}(\Sigma, \mathcal{V})$ such that $(t, r) \in \rightarrow^+_{\mathcal{R}}$, and we denote it as $t \downarrow_{\mathcal{R}}$. Term t* ***is a normal form of*** *s if $(s, t) \in \rightarrow^*_{\mathcal{R}}$ and t is a normal form; we denote it as $t \downarrow_{\mathcal{R}} s$.*

Example 2.1.19 (Term rewriting system) *Now we can develop the rewriting system described in Example 2.1.4. In this rewriting system we have a signature $\Sigma = \{zero/0, true/0, false/0, s/1, not/1, add/2, mult/2, leq/2\}$, and now we define the set of rules R that makes the system behave as expected, considering the intuitive meaning of the symbols:*

$$
\begin{array}{lcl}
not(true) & \rightarrow & false \\
not(false) & \rightarrow & true \\
 & & \\
add(X, zero) & \rightarrow & X \\
add(X, s(Y)) & \rightarrow & s(add(X, Y)) \\
 & & \\
mult(X, zero) & \rightarrow & zero \\
mult(X, s(Y)) & \rightarrow & add(X, mult(X, Y)) \\
 & & \\
leq(zero, X) & \rightarrow & true \\
leq(s(X), zero) & \rightarrow & false \\
leq(s(X), s(Y)) & \rightarrow & leq(X, Y)
\end{array}
$$

There are no rules that have a term with a symbol $zero$, $true$, $false$ or s at the head of the left-hand side of them. These symbols that have no rules for them are usually called ***data constructor symbols*** *and represent data elements in the rewriting system; the other symbols that have rewrite rules defined for them represent* ***defined function symbols*** *and can be applied to terms in order to obtain a different term as defined in the reduction relation (see Definition 2.1.17).*

Now we show some reduction sequences, using the notation $s \rightarrow_r^\sigma t$ *to mean that* s *reduces to* t *making explicit the substitution* σ *and the rule* r *applied in the rewriting step. Notice that we employ variants of the rewrite rules formed by adding a subscript corresponding to the order of the rewriting step in the reduction sequence to the variables in the rule to avoid clashes with the names of the variables in the terms. We also underline the subterm where the rewrite step is applied.*

We show now a rewriting sequence to prove that $(mult(s(zero), add(s(zero), zero)), s(zero)) \in \rightarrow_{\mathcal{R}}^*$ *(i.e.,* $1 * (1 + 0) = 1$*):*

$$
\begin{aligned}
& mult(s(zero), \underline{add(s(zero), zero)}) && \rightarrow^{\{X_1 \mapsto s(zero)\}}_{add(X_1, zero) \rightarrow X_1} \\
& \underline{mult(s(zero), s(zero))} && \rightarrow^{\{X_2 \mapsto s(zero), Y_2 \mapsto zero\}}_{mult(X_2, s(Y_2)) \rightarrow add(X_2, mult(X_2, Y_2))} \\
& add(s(zero), \underline{mult(s(zero), zero)}) && \rightarrow^{\{X_3 \mapsto zero\}}_{mult(X_3, zero) \rightarrow zero} \\
& \underline{add(s(zero), zero)} && \rightarrow^{\{X_4 \mapsto s(zero)\}}_{add(X_4, zero) \rightarrow X_4} \\
& s(zero)
\end{aligned}
$$

We have proved that $mult(s(zero), add(s(zero), zero))$ *reduces to* $s(zero)$*. Since* $s(zero)$ *is a normal form, because is formed only by data constructor symbols and can not be applied any rewrite rule to it, we have that* $s(zero)$ *is a normal form of* $mult(s(zero), add(s(zero), zero))$*.*

We show now another reduction sequence in order to prove that $(not(leq(add(zero, X), s(zero))), true) \in \rightarrow_{\mathcal{R}}^*$*, that considering the pretended meaning of the symbols is proving that* $\neg(0 + x \leq 1)$ *(i.e.,* $0 + x > 1$*):*

$$
\begin{aligned}
& not(leq(\underline{add(zero, X)}, s(zero))) && \rightarrow^{\{X \mapsto s(Y_1), X_1 \mapsto zero\}}_{add(X_1, s(Y_1)) \rightarrow s(add(X_1, Y_1))} \\
& not(leq(s(\underline{add(zero, Y_1)}), s(zero))) && \rightarrow^{\{X_2 \mapsto zero, Y_1 \mapsto s(Y_2)\}}_{add(X_2, s(Y_2)) \rightarrow s(add(X_2, Y_2))} \\
& not(leq(s(s(\underline{add(zero, Y_2)})), s(zero))) && \rightarrow^{\{X_3 \mapsto zero, Y_2 \mapsto zero\}}_{add(X_3, zero) \rightarrow X_3} \\
& not(\underline{leq(s(s(zero)), s(zero))}) && \rightarrow^{\{X_4 \mapsto s(s(zero)), Y_4 \mapsto s(zero)\}}_{leq(s(X_4), s(Y_4)) \rightarrow leq(X_4, Y_4)} \\
& not(\underline{leq(s(zero), zero)}) && \rightarrow^{\{X_5 \mapsto zero\}}_{leq(s(X_5), zero) \rightarrow false}
\end{aligned}
$$

$$\underline{not(false)} \qquad \rightarrow^{\varepsilon}_{not(false)\rightarrow true}$$

$$true$$

In some steps there is more than one rule that matches with the current term, and in any implementation of a reduction process must be defined the way that the rules are chosen: this may be a fixed way, for example, choosing always the rule that appears first in the rewriting system, or even choosing one at random. Now we present another reduction sequence for the same starting term of the previous sequence:

$$not(leq(\underline{add(zero, X)}, s(zero))) \quad \rightarrow^{\{X\mapsto zero, X_1\mapsto zero\}}_{add(X_1, zero)\rightarrow X_1}$$

$$not(\underline{leq(zero, s(zero))}) \qquad \rightarrow^{\{X_2\mapsto s(zero)\}}_{leq(zero, X_2)\rightarrow true}$$

$$\underline{not(true)} \qquad \rightarrow^{\varepsilon}_{not(true)\rightarrow false}$$

$$false$$

As we can see from these sequences, any given term may be reduced to two or more different normal forms; when defining a programming language implementing a reduction strategy, it can be chosen a strategy that guarantees that only one answer is computed or all the answers can be computed. In the former is usually desirable that the same answer is always computed to guarantee the capability of reproducing the computations; in that case, the language is said to be deterministic.

Two very interesting properties of term rewriting systems are the properties of *termination* and *confluence*. The first one is fulfilled by the rewriting system if all reduction sequences starting in any term are finite, i.e., eventually reach a normal form. The confluence property is verified if when it is possible to reduce a given term to two different ones, both of them can be reduced to a common term. In a confluent and terminating rewriting system holds that each term has an unique normal form. Termination and confluence are undecidable properties, but there has been developed methods that allow to prove them for many rewriting systems. These methods are out of the scope of this work since we do not impose any restriction about confluence termination in our framework; additional information about these two properties can be found in [5, 8, 11].

Term rewriting systems as described in this section are not able to deal with *higher-order functions* (i.e., functions that can have functions as arguments and/or results), that is the goal of this work. The main problem here is that we would need the capacity to use function symbols as variables. One classical way of defining higher-order term rewriting systems is by means of λ-*calculus* [8, 67], that is

explained in the next section.

2.2 The λ-Calculus

The λ-calculus is a family of prototype programming languages developed by Alonzo Church in the 1930's [19]. Their main feature is that they are *higher-order*; that means that they provide operators that can have other operators as input and/or output. Although the syntax of this language is very simple and reduced, it is another Turing-complete computational model that can describe the same computations as the most commonly used programming languages (see e.g., [32]).

The λ-calculus can be studied as a term rewriting system, in a context where terms are called *λ-terms* and rewrite rules are three *conversion rules* (also called *reduction rules*). Even though, we need to adapt in the context of λ-calculus some of the notions formulated in the previous section for term rewriting systems in order to use the most common notations when referring to each formalism. In the following section, where two formalisms are put together, we clarify the notations used once referring to concepts influenced by both of them.

In this section we study first the *type-free λ-calculus*, which is the most simple language of the λ-calculus family; in spite of its simplicity, most of the concepts that we need for our framework regarding to λ-calculus can be defined in this level. After that we give some notions of the *simply-typed λ-calculus*, adding types to the language defined before, that is essential to define the syntactic level of the programming framework. There are many excellent references covering λ-calculus from many points of view. An encyclopedic volume on λ-calculus, where the simply typed version is barely sketched is [6]; a nice more didactic book devoted to computer scientists is [43]; finally, a more type focused book is [51].

2.2.1 Type-Free λ-Calculus

Type-free λ-calculus is a very simple formalism whose syntactic constructions are λ-terms. The most basic λ-terms are defined from a set of variables; then additional terms are built by means of two constructions: the first one, called *abstraction*, intuitively corresponds to a procedure in a programming language, allowing a variable in a term to be a parameter that can be instantiated by another term; the second one, called *application*, corresponds to the application of one term to another, in an analogous way to calling a procedure with a parameter instantiated.

Definition 2.2.1 (λ-terms) *Let $\mathcal{V}$ be a set of variables. The set of* ***λ-terms*** *defined over $\mathcal{V}$ is denoted as $\Lambda(\mathcal{V})$ (or simply Λ) and is defined inductively as follows:*

- $\forall v \in \mathcal{V}.\ v \in \Lambda$. *These terms are called* ***atomic terms*** *or simply* ***atoms****.*

- $\forall t \in \Lambda, v \in \mathcal{V}.\ \lambda v.t \in \Lambda$. *These terms are called* λ***-abstractions***.
- $\forall t_1, t_2 \in \Lambda.\ (t_1 t_2) \in \Lambda$. *These terms are called* ***applications***.

We assume that application is left-associative, abstraction is right-associative and the most external pair of parenthesis can be omitted (i.e., $\lambda x.\lambda y.xy$ represents the term $\lambda x.(\lambda y.(xy))$ and xyz represents the term $((xy)z)$), and application has higher priority than abstraction (i.e., $\lambda x.xy$ represents $\lambda x.(xy)$ and not $(\lambda x.x)y$).

By means of this definition, variables appearing in any given term can be classified into two sets: the one with variables that are abstracted, called *bound variables* and the ones that are not, called *free variables*. These sets of variables are easily defined by means of intuitive recursive definitions:

Definition 2.2.2 (Bound and free variables) *Let* $t \in \Lambda$.

1. *The set of* ***bound variables*** *of* t, *denoted as* $\mathcal{BV}(t)$, *is defined by induction over the structure of* t *as follows:*
   - *if* t *is an atom, then* $\mathcal{BV}(t) = \emptyset$.
   - *if* $v \in \mathcal{V}$, $s \in \Lambda$, *and* $t = \lambda v.s$, *then* $\mathcal{BV}(t) = \{v\} \cup \mathcal{BV}(s)$.
   - *if* $r, s \in \Lambda$ *and* $t = rs$ *then* $\mathcal{BV}(t) = \mathcal{BV}(r) \cup \mathcal{BV}(s)$.
2. *The set of* ***free variables*** *of* t, *denoted as* $\mathcal{FV}(t)$, *is defined by induction over the structure of* t *as follows:*
   - *if* t *is an atom, then* $\mathcal{FV}(t) = \{t\}$.
   - *if* $v \in \mathcal{V}$, $s \in \Lambda$, *and* $t = \lambda v.s$, *then* $\mathcal{FV}(t) = \mathcal{FV}(s) \backslash \{v\}$.
   - *if* $r, s \in \Lambda$ *and* $t = rs$ *then* $\mathcal{FV}(t) = \mathcal{FV}(r) \cup \mathcal{FV}(s)$.
3. *Term* t *has a* ***bound-variable clash*** *if there exists* $v \in Var(t)$ *such that* $v \in \mathcal{BV}(t)$ *and* $v \in \mathcal{FV}(t)$.
4. *Term* t *is a* ***ground term*** *(or* ***combinator****) if* $\mathcal{FV}(t) = \emptyset$.

It is also useful to formally define the concept of *subterm* when reasoning about λ-terms, that is very similar to the one given for generic term rewriting systems (see Definition 2.1.8).

Definition 2.2.3 (Subterms) *Let* $t \in \Lambda$. *The set of* ***subterms*** *of* t, *denoted as* $Subterm(t)$, *is defined by induction over the structure of* t *as follows:*

- *if* t *is an atom, then* $Subterm(t) = \{t\}$.

- *if* $v \in \mathcal{V}$, $s \in \Lambda$, *and* $t = \lambda v.s$, *then* $Subterm(t) = \{t\} \cup Subterm(s)$.
- *if* $r, s \in \Lambda$, *and* $t = rs$, *then* $Subterm(t) = \{t\} \cup Subterm(r) \cup Subterm(s)$.

There is some specific terminology when referring to λ-abstractions $t = \lambda v.s$: the bound variable v is called the *abstracted variable* and the subterm s where that variable is abstracted by means of the corresponding abstraction is called the *scope* of the abstracted variable v or the *body* of the abstraction t.

Example 2.2.4 (Syntax of λ-terms) *In this example we define some simple λ-terms to illustrate some of the concepts already defined in this section; they are not intended to have any particular intuitive meaning. In the next section we provide more interesting examples of λ-terms with an intended meaning, thanks to the extensions to type-free λ-calculus that provide the framework that is the object of study of our work.*

We consider a set of variables $\mathcal{V} = \{x, y, z\}$. *Some λ-terms in* $\Lambda(\mathcal{V})$ *are:*

- $t_1 \equiv x$ *is an atom, such that* $\mathcal{BV}(t_1) = \emptyset$, $\mathcal{FV}(t_1) = \{x\}$, *and* $Subterm(t_1) = \{x\}$.
- $t_2 \equiv xy$ *is an application, such that* $\mathcal{BV}(t_2) = \emptyset$, $\mathcal{FV}(t_2) = \{x, y\}$, *and* $Subterm(t_2) = \{x, y, xy\}$.
- $t_3 \equiv \lambda x.(xy)(xy)$ *is a λ-abstraction where x is the abstracted variable and $(xy)(xy)$ is the scope of x and the body of the abstraction.* $\mathcal{BV}(t_3) = \{x\}$, $\mathcal{FV}(t_3) = \{y\}$, *and* $Subterm(t_3) = \{x, y, xy, (xy)(xy), \lambda x.(xy)(xy)\}$.

In order to define the reduction notions in this framework, it is needed to adapt the concept of *substitution*. Here the definition is more complicated than in term rewriting systems, because of the existence of λ-abstractions. An abstraction makes a variable to be bound; in this case the variable is just a name and changing it by another name that does not appear in the body of the abstraction would not have to make the abstraction to behave in a different way. So that, in this context substitutions only change the value of free variables, and another operation is necessary for changing the name of bound variables. In this definition taken from [43], and in contrast to the analogous one for term rewriting systems, substitutions only change the value of one variable by a different term.

Definition 2.2.5 (Substitution) *Let* $x \in \mathcal{V}$, $s, t \in \Lambda$. *The* ***substitution*** *of the variable x by s in t is denoted as* $t[x/s]$ *and is recursively defined as follows:*

1. *if* $t = x$ *then* $t[x/s] = s$.

2. *if* $t \in \mathcal{V}$, *and* $t \neq x$ *then* $t[x/s] = t$.
3. *if* $r_1, r_2 \in \Lambda$, *and* $t = r_1 r_2$ *then* $t[x/s] = r_1[x/s]r_2[x/s]$.
4. *if* $r \in \Lambda$, *and* $t = \lambda x.r$ *then* $t[x/s] = \lambda x.r$.
5. *if* $r \in \Lambda$, $y \in \mathcal{V}$, $x \neq y$, $t = \lambda y.r$, *and* $x \notin \mathcal{FV}(r)$ *then* $t[x/s] = \lambda y.r$.
6. *if* $r \in \Lambda$, $y \in \mathcal{V}$, $x \neq y$, $t = \lambda y.r$, $x \in \mathcal{FV}(r)$, *and* $y \notin \mathcal{FV}(s)$ *then* $t[x/s] = \lambda y.r[x/s]$.
7. *if* $r \in \Lambda$, $y, z \in \mathcal{V}$, $x \neq y \neq z$, $t = \lambda y.r$, $x \in \mathcal{FV}(r)$, *and* $y \in \mathcal{FV}(s)$ *then* $t[x/s] = \lambda z.r[y/z][x/s]$.

A notation that is useful is the one of *simultaneous substitution*. This consists simply in applying at the same time some substitutions to variables that appear in a given term, and renaming variables when needed in order to avoid variable clashes according to the cases defined in Definition 2.2.5. For example, the simultaneous substitution $t[x_1/t_1, \ldots, x_n/t_n]$ represents the sequential substitution of the occurrences of x_1 by t_1, ..., x_n by t_n in t avoiding clashes of variables.

Example 2.2.6 (Substitutions) *In this example we apply substitutions to terms in* $\Lambda(\{x, y, z, t\})$, *each one corresponding to one case of those defined in Definition 2.2.5.*

1. $x[y/yy] = x$.
2. $x[x/xx] = xx$.
3. $xy[x/xx] = x[x/xx]y[x/xx] = xx(y[x/xx]) = (xx)y$.
4. $\lambda x.xy[x/xx] = \lambda x.xy$.
5. $\lambda x.x(\lambda y.xy)[y/xy] = \lambda x.x(\lambda y.xy)$.
6. $(\lambda x.xy)[y/zz] = \lambda x.x(zz)$.
7. $(\lambda x.xy)[y/xx] = \lambda z.xy[x/z][y/xx] = \lambda z.zy[y/xx] = \lambda z.z(xx)$.

Finally, an example of simultaneous substitution is $x(y\lambda z.(xy)z)[x/t, y/tt, z/ttt] = t(tt\lambda z.(t(tt))z)$.

Sometimes it is useful to change the name of an abstracted variable in a given λ-term for another name. This process, called *α-conversion*, is one of the reduction rules of the λ-calculus, and can be performed only in the case that the new name does not occur in the body of the λ-abstraction. Working with terms that have an

abstracted variable that occurs as a free one out of the abstraction (for example x in $x(\lambda x.xy)$) presents some difficulties when working with the other reduction rules of the paradigm that are defined later; a solution to this problem is changing the name of bound variables to others that do not appear as free variables in the term when needed, by means of the α-conversion equivalence. From now on, it is assumed that α-conversion is applied when needed to avoid variables clashes.

Definition 2.2.7 (α-conversion) *Let $t \in \Lambda$, $x, y \in \mathcal{V}$, and $y \notin \mathcal{FV}(t)$. Then we say that $\lambda x.t$ is reduced to $\lambda y.t[x/y]$ by α-**conversion** and is denoted as $\lambda x.t \equiv_\alpha \lambda y.t[x/y]$. If $s \in \Lambda$ can be reduced to t performing zero or more number of α-conversion steps, we say that s α-**converts** to t and we denote it as $s \equiv_\alpha t$.*

In general, the α-conversion reduction rule is non-terminating and infinite steps renaming bound variables can be applied to any term.

Example 2.2.8 (α-conversion) *In this example we apply α-conversion to different terms in $\Lambda(\{x, y, z\})$:*

- *We have that $\lambda x.x(\lambda y.xy) \equiv_\alpha \lambda z.z(\lambda y.zy)$ by changing the variable x to z but we cannot rename x to y because $y \in \mathcal{FV}(t)$.*
- *To avoid a variable clash of the variable x in $t \equiv x(\lambda x.xy)$ we can reduce it by α-conversion: $x(\lambda x.xy) \equiv_\alpha x(\lambda z.zy)$. Now x appears in t only as a free variable because the bound variable name was changed to z.*

The α-conversion reduction rule defined before is one of the three reduction rules that are defined into the λ-calculus. The other two are *β-reduction* and *η-reduction*. The first one reduces abstractions and applications; if we consider abstractions as functions with a parameter (being the abstracted variable the parameter of the function), the β-conversion reduction rule is analogous to the process of passing a parameter to the function, instantiating all the occurrences of the abstracted variable in the abstraction by the parameter passed by means of an application. A term is said to be in *β-normal form* if and only if no β-conversion steps can be performed from it; in other words, it has no subterms of the form $(\lambda x.t)s$.

Definition 2.2.9 (β-reduction) *Let $s, t \in \Lambda$ and $x \in \mathcal{V}$. The term $(\lambda x.t)s$ is a β-**redex**. Its **contractum** is the term $t[x/s]$. The β-**reduction** of t is the evaluation of zero or more β-redexes in t to its corresponding contractums. The term t β-**reduces** to s for each term s reached in the process and it is denoted as $t \to_\beta s$.*

Definition 2.2.10 (β-normal form) *Let s, t in Λ. Term t is in β-**normal form** if t does not have a subterm that is a β-redex. Term t is a β-**normal form** of s if $s \to_\beta t$ and t is a β-normal form.*

The β-reduction rule is confluent but non-terminating. So that, not every λ-term has a β-normal form, but in the case it has one then it is unique (see [6, 43] for additional details). There are reduction strategies to compute the β-normal form of a term in the case it has one that can be found in the mentioned references.

Example 2.2.11 (β-reduction) *In this example we apply β-reduction to different terms in $\Lambda(\{x, y, z\})$:*

- $(\lambda x.x(\lambda y.xy)z)z \rightarrow_\beta (\lambda x.(x(xz))z)z \rightarrow_\beta (z(zz))z$. *We have that $(z(zz))z$ is a β-normal form because it does not have any β-redexes, and is a β-normal form of $(\lambda x.x(\lambda y.xy)z)z$, $(\lambda x.(x(xz))z)z$, and itself.*
- $(\lambda x.y(xx))(\lambda x.y(xx)) \rightarrow_\beta y((\lambda x.y(xx))(\lambda x.y(xx))) \rightarrow_\beta y(y((\lambda x.y(xx))(\lambda x.y(xx)$ $))) \rightarrow_\beta \cdots$. *In this case $(\lambda x.y(xx))(\lambda x.y(xx))$ does not have a β-normal form.*

The third reduction rule of the λ-calculus framework is called *η-reduction*. This rule has an analogy with the notion of extensional equality of functions. This notion states that two functions are equal if and only if for all possible input arguments both return the same output. In contrast to the β-reduction rule defined before, η-reductions always terminate and every term has an *η-normal form*; this is obvious since in every η-reduction step an η-redex is consumed.

Definition 2.2.12 (η-reduction) *Let $s, t \in \Lambda$, $x \in \mathcal{V}$ and $x \notin \mathcal{FV}(t)$. The term $\lambda x.tx$ is an η-**redex**. Its **contractum** is the term t. The η-**reduction** of t is the evaluation of zero or more η-redexes in t to its corresponding contractum. The term t η-**reduces** to s for each term s reached in the process and it is denoted as $t \rightarrow_\eta s$.*

Definition 2.2.13 (η-normal form) *Let s, t in Λ. Term t is in η-**normal form** if t does not have a subterm that is an η-redex. Term t is an η-**normal form** of s if $s \rightarrow_\eta t$ and t is an η-normal form.*

Example 2.2.14 (η-reduction) *Term $\lambda x.(\lambda y.zzy)x$ can be η-reduced, as shown by the following reduction sequence: $\lambda x.(\lambda y.zzy)x \rightarrow_\eta \lambda x.zzx \rightarrow_\eta zz$. We have that zz is an η-normal because it does not have any η-redexes, and is an η-normal form of $\lambda x.(\lambda y.zzy)x$, $\lambda x.zzx$, and itself.*

The last two reduction rules defined before can be combined to define another reduction rule, called *$\beta\eta$-conversion* and defined as follows:

Definition 2.2.15 ($\beta\eta$-reduction) *Let $s, t \in \Lambda$. The term t is a $\beta\eta$-**redex** if it is a β-redex or it is an η-redex. Its **contractum** is the corresponding one to the redex. The $\beta\eta$-**reduction** of t is the evaluation of zero or more $\beta\eta$-redexes in t to its corresponding contractums. The term t $\beta\eta$-**reduces** to s for each term s reached in the process and it is denoted as $t \rightarrow_{\beta\eta} s$.*

Since any term is not guaranteed to have a β-normal form, it is not guaranteed to have a $\beta\eta$-normal form. In the next section we define a *type system* for λ-calculus that rejects every term that cannot be $\beta\eta$-reduced.

Definition 2.2.16 ($\beta\eta$-normal form) *Let s, t in Λ. Term t is in $\beta\eta$-**normal form** if t does not have a subterm that is a $\beta\eta$-redex. Term t is a $\beta\eta$-**normal form** of s if $s \to_{\beta\eta} t$ and t is a $\beta\eta$-normal form.*

Example 2.2.17 ($\beta\eta$-reduction) *Term $(\lambda x.(\lambda y.yy)z)x$ can be $\beta\eta$-reduced, as is shown by the following reduction sequence: $(\lambda x.(\lambda y.yy)z)x \to_{\beta\eta} \lambda x.zzx \to_{\beta\eta} zz$. We have that zz is a $\beta\eta$-normal form because it does not have any $\beta\eta$-redexes, and is a $\beta\eta$-normal form of $(\lambda x.(\lambda y.yy)z)x$, $\lambda x.zzx$, and itself.*

Finally we provide an important concept for our framework, the *long $\beta\eta$-normal form* of a term. The name is misleading, since a term in long $\beta\eta$-normal form is not necessarily in $\beta\eta$-normal form, the long $\beta\eta$-normal form of a term t is the η-extended form of the β-normal form of t. The concept of long $\beta\eta$-normal form is defined by means of the auxiliary notion of η-expansion:

Definition 2.2.18 (Long $\beta\eta$-normal form)

- *Let $t \in T(\Sigma_\perp, \mathcal{V})$ be in β-normal form such that $t = \lambda\overline{x_n}.a(\overline{t_m})$. The η-**expanded form** of t is represented as $t\uparrow_\eta$ and defined as:*

$$t\uparrow_\eta = \lambda\overline{x_{n+k}}.a(\overline{t_m}\uparrow_\eta, \overline{x_{n+k}}\uparrow_\eta)$$

  such that $\overline{x_{n+k}} \notin \mathcal{FV}(\overline{t_m})$ and $t :: \overline{\tau_{n+k}} \to \tau$.

- *Let $t \in T(\Sigma_\perp, \mathcal{V})$. The **long** $\beta\eta$-**normal form** of t is defined as $t\updownarrow^\eta_\beta = (t\downarrow_\beta)\uparrow_\eta$, that is, the η-expanded form of the β normal form of t.*

It is well-known that $s \to_{\alpha\beta\eta} t$ if $s\updownarrow^\eta_\beta \equiv_\alpha t\updownarrow^\eta_\beta$ for any given terms s and t (see [52]).

2.2.2 Simply-Typed λ-Calculus

In this section we present an extension to type-free λ-calculus adding a *type system* to it. A type system is a syntactic method of classifying terms in a language according to the kind of values they compute, effectively assigning a *type* to them; it is possible to have terms that do not have a type assigned under the type system, and usually correspond to terms that would present some kind of undesired behavior when reduced. In modern programming languages supporting types, it is usual to have valid chains of symbols considering the lexical and the syntax of terms in the

language, but rejected by the type system at the semantic level so the operational mechanism does not deal with them. Type checking at compile time has been an invaluable aid to detect errors for the programmer of programming languages supporting it. A very complete reference about types in programming languages with an specific chapter devoted to simply-typed λ-calculus is [79].

Simply-typed λ-calculus is an extension to the type-free version by assigning types to the terms of the language. Types are built from a set of *base types* (e.g., *int*, *nat*, *bool*, *real*, ...) and a *type constructor* represented by the operator $\rightarrow$ (that associates to the right) to represent functions; a type $\tau_1 \rightarrow \tau_2$ represents a function that receives a term with type τ_1 as input and returns a value with type τ_2. Not every term of the language has a type under the type system we are defining. In this type system variables are expected to have a type explicitly assigned, abstractions to have a functional type, and applications the type of the result of applying its first argument to a term of the type of its second argument. Usually, the syntax of the language is modified in order to add explicitly the type of variables (see [79]); in this work we assume the existence of a *type environment* Γ that contains this information. A type environment is a mapping from the variables in the term to the set of valid types; assuming that we avoid bound variable clashes and that variables in different abstractions are kept disjoint by means of α-conversion, then the type environment is the same when computing types at any position of the term. We represent the type environment as a list of *type assumptions* with the syntax $v : T$ that states that variable v has type T.

Definition 2.2.19 (Valid Types) *Let $\mathcal{B}$ be a set of base types. The valid **types** over $\mathcal{V}$ are the following:*

- *if $\tau \in \mathcal{B}$ then τ is a valid type. These types are called **atomic types**.*
- *if τ_1 and τ_2 are valid types then $\tau_1 \rightarrow \tau_2$ is a valid type. These types are called **composite types** or **functional types**.*

Example 2.2.20 (Typed terms) *In this example we provide the intuitive type of some terms, before defining the type system that induces an algorithmic procedure to compute the type of any term that has a type. We assume a set of base types $\mathcal{B} = \{int, bool\}$. We use the notation $t :: T$ to represent that term t has type T. In this language, the type of variables must be explicitly provided when defining the terms.*

- *$x :: bool$ with $\Gamma = [x : bool]$. In this example we have a term consisting of a single variable, and the type of single variables have to be provided under this type system by the type environment with our assumptions.*

- *$fx :: int$ with $\Gamma = [f : bool \to int, x : bool]$. The application of one term to another expects that the first term is a functional term and the latter has the type of the argument the function is expecting; then the type of the application is the one that the functional argument returns.*

- *$\lambda x.x :: int \to int$ with $\Gamma = [x : int]$. Abstractions represent functions that have as first argument a term of the type of the abstracted variable and return a term with the same type as the body of the abstraction.*

- *$\lambda x.xx$ does not have a type assigned. In this term, variable x is applied to a term, so it must have a functional type $T_1 \to T_2$, but is also its first argument, so it must have a type T_1, that must be of the form $T_1' \to T_2' \ldots$ This would lead to an endless argument and the conclusion is that under this type system the term does not have a type assigned.*

The type system defined here is *monomorphic*, because every variable must have a type assigned to it; a more expressive language can be defined using *polymorphic types*; in such a language, a variable is not assigned a fixed type but a variable type by the type system, and terms in the language are assigned the most general type that can be evaluated constrained by their definition. For example, term $\lambda x.x$ would be assigned a type $\alpha \to \alpha$ under a polymorphic type system, being α a *type variable*, and represents the identity function of any type; under a monomorphic type system we would have an identity function for each kind of arguments we are going to use it ($\lambda x.x :: int \to int$ with $\Gamma = [x : int]$, $\lambda x.x :: bool \to bool$ with $\Gamma = [x : bool]$, ...). Polymorphic type systems are more expressive than monomorphic ones, but also make the theoretical development more complex and would not add computational capacity to the framework, so we have decided to use a monomorphic type system when defining our framework in the next section. Polymorphic type systems for λ-calculus can be found in [43, 79].

In Figure 2.1 we define the typing rules for simply typed λ-calculus adapted from [79]. With these rules we can derive statements of the form $\Gamma \vdash t :: T$ that means that term t has type T under an environment Γ. We assume that the type environment is initialized with the correct type statements.

Example 2.2.21 (Type derivations) *This example shows how we can derive the type $int \to bool \to int$ of the expression $\lambda x.\lambda y.fxy$ under an environment $\Gamma = [x : int, y : bool, f : int \to bool \to int]$ applying the rules of the calculus defined in Figure 2.1.*

$$\dfrac{\dfrac{\dfrac{}{\Gamma(x) = int}}{\Gamma \vdash x :: int}\ \textbf{VAR}}{\Gamma \vdash \lambda x.\lambda y.fxy :: int \to bool \to int}\ \textbf{ABS}$$

VAR VARiables	$\dfrac{\Gamma(x) = T}{\Gamma \vdash x :: T}$
ABS ABStractions	$\dfrac{\Gamma \vdash x :: T_1 \qquad \Gamma \vdash t :: T_2}{\Gamma \vdash \lambda x.t :: T_1 \to T_2}$
APP APPlications	$\dfrac{\Gamma \vdash t_1 :: T_1 \to T_2 \qquad \Gamma \vdash t_2 :: T_1}{\Gamma \vdash t_1\, t_2 :: T_2}$

Figure 2.1: The typing rules for the simply-typed λ-calculus.

$$
\begin{array}{l}
\textbf{ABS}\ \Gamma \vdash \lambda y.fxy :: \mathit{bool} \to \mathit{int} \\
\quad \textbf{VAR}\ \Gamma \vdash y :: \mathit{bool} \\
\qquad \Gamma(y) = \mathit{bool} \\
\quad \textbf{APP}\ \Gamma \vdash fxy :: \mathit{int} \\
\qquad \textbf{APP}\ \Gamma \vdash fx :: \mathit{bool} \to \mathit{int} \\
\qquad\quad \textbf{VAR}\ \Gamma \vdash f :: \mathit{int} \to \mathit{bool} \to \mathit{int} \\
\qquad\qquad \Gamma(f) = \mathit{int} \to \mathit{bool} \to \mathit{int} \\
\qquad\quad \textbf{VAR}\ \Gamma \vdash x :: \mathit{int} \\
\qquad\qquad \Gamma(x) = \mathit{int} \\
\qquad \textbf{VAR}\ \Gamma \vdash y :: \mathit{bool} \\
\qquad\quad \Gamma(y) = \mathit{bool}
\end{array}
$$

An essential difference between type-free λ-calculus and simply typed λ-calculus is that β-reduction is terminating, and every term has an unique βη-normal form (see a formal proof of this non-trivial result in [43]), that is key in our framework. The reason for this is that terms that are problematic for β-reduction, that are terms where a variable is replicated and self-applied (such as $\lambda x.xx$, see Example 2.2.20), are rejected by the type system. An immediate consequence of this is that simply typed λ-calculus is not a Turing-complete computational model with the reduction relations defined for λ-calculus, because computational models where all computations terminate are not Turing-complete (see [21, 32]). This does not mean that our framework is not Turing-complete, since we only use simply-typed λ-calculus at the syntactic level and the reduction relation is defined analogously to the one defined for term rewriting systems.

2.3 Higher-Order Pattern Rewriting Systems

In this section we define the syntax and some basic concepts of the language that is the object of study of this work. Most of the definitions in this section are adaptations or simple extensions of the ones given for term rewriting systems and λ-calculus. This framework is a higher-order extension based on λ-calculus to term rewriting systems. The original idea of using patterns to add higher-order capacities to declarative programming languages can be found in [70]; a complete modern research of pattern rewriting systems is [28]; finally, its application in a functional logic framework can be found in [81].

In this framework, basic terms are elements from a set of function symbols and a set of variables; complex ones are constructed by means of composition and abstraction of more simple terms. All terms in our system have a type assigned from a set of types in a similar fashion than we defined for simply typed λ-calculus; also we need to add for semantic purposes to the set of terms a special constant for each of the *base types* of the system; these constants are called the *bottom* constant or simply *bottom* and are denoted by the symbol $\bot$, possibly subscripted by the name of the type when it is not obvious from the context ($\bot_b, \forall b \in \mathcal{B}$). The bottom value represents an *undefined* value for each base type and is a key feature of our framework; its relevance will manifest in the following chapters of this work and its role will be more clear when defining the basic semantic notions relating terms later in this section. Bottom constants are not allowed to appear in programs and in the operational semantics they represent computations that are not performed in order to evaluate goals from programs. A term that does not have a constant $\bot$ is called a *total term*, and when referring to the set of them we use the notation $T(\Sigma, \mathcal{V})$; those terms that might have at least one bottom constant at any position are called *partial terms*, and when referring to the set of terms including them we use the notation $T(\Sigma_\bot, \mathcal{V})$.

Definition 2.3.1 (Terms) *Let Σ be a signature, $\mathcal{B}$ a set of base types, and $\mathcal{V}$ a set of variables, such that $\Sigma \cap \mathcal{V} = \emptyset$. The set of all* ***$\lambda$-terms*** *over $\mathcal{V}$ is denoted as $T(\Sigma_\bot, \mathcal{V})$, and is defined inductively as follows:*

- $\forall f \in \Sigma.\ f \in T(\Sigma_\bot, \mathcal{V})$.
- $\forall v \in \mathcal{V}.\ v \in T(\Sigma_\bot, \mathcal{V})$.
- $\forall b \in \mathcal{B}.\ \bot_b \in T(\Sigma_\bot, \mathcal{V})$. *The set of these terms $\bot_b$ for each base type b of $\mathcal{B}$ is denoted as Bot, and $\Sigma \cup Bot$ is denoted as $\Sigma_\bot$.*
- $\forall t_1, t_2 \in T(\Sigma_\bot, \mathcal{V}).\ (t_1 t_2) \in T(\Sigma_\bot, \mathcal{V})$.
- $\forall t \in T(\Sigma_\bot, \mathcal{V}), x \in \mathcal{V}.\ \lambda x.t \in T(\Sigma_\bot, \mathcal{V})$.

For brevity and to improve readability, we define a notation that will be used in the rest of this work: a sequence of syntactic objects $x_1, x_2, \ldots, x_n$, where $n \geq 0$, is denoted as $\overline{x_n}$. Successive applications of terms $(((a\, t_1) t_2) \cdots t_n)$ are denoted as $a(t_1, t_2, \ldots, t_n)$ or $a(\overline{t_n})$, where a is a function symbol such that $\mathit{arity}(a) \geq n$ or a variable, and successive abstraction of variables $\lambda x_1.\lambda x_2.\cdots.\lambda x_n.t$ are denoted as $\lambda x_1, x_2, \cdots, x_n.t$ or $\lambda\overline{x_n}.t$.

All terms have assigned a type, with the same rules defined in the context of the simply-typed λ-calculus (see again Figure 2.1). Terms that do not have a type assigned under this type system are out of the framework. Because of this, every term is guaranteed to have a $\beta\eta$-normal form and consequently a long $\beta\eta$-normal form, which is assumed to be an implicit operation (i.e., every term is expected to be normalized with respect the long $\beta\eta$-relation at any point). We also assume that terms are identified modulo α-conversion, and in general we avoid bound-variable clashes when manipulating terms, keeping free and bound variables disjoint using implicit α-conversion operations when needed, and also assuming that bound variables with different binders have different names. For brevity, we may write variables and constants from Σ in η-normal form (i.e., X instead of $\lambda\overline{x_k}.\, X(\overline{x_k})$). With these conventions, every considered term t in this framework has an unique long $\beta\eta$-normal form $\lambda\overline{x_k}.\, a(\overline{t_n})$, where $a \in \Sigma_\perp \cup \mathcal{V}$ and $a()$ coincides with a. We refer to symbol a as the *head* or *root* of the term and denote it as $hd(t)$.

For semantic purposes, it is useful to consider the set of (possibly) partial terms $T(\Sigma_\perp, \mathcal{V})$ as a partial ordered set (poset) with bottom (see Section 4.1.1) with respect to an *approximation ordering* $\sqsubseteq$, where lesser terms in the order are elements with less information (less symbols distinct of $\perp$), the bottom constant is the minimum element, and total terms are maximal elements; thus, is defined as the least partial ordering such that:

$$\perp \sqsubseteq t \qquad t \sqsubseteq t \qquad \frac{s_1 \sqsubseteq t_1 \ \cdots \ s_n \sqsubseteq t_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \sqsubseteq \lambda\overline{x_k}.\, a(\overline{t_n})}$$

Example 2.3.2 (Terms) *In this section we develop a higher-order example with some basic operations on the Peano natural numbers. We consider* $\Sigma = \{zero/0 : nat, s/1 : nat \to nat, add/2 : nat \to nat \to nat, mult/2 : nat \to nat \to nat, diff/2 : (nat \to nat) \to nat \to nat\}$ *and* $\mathcal{B} = \{nat\}$*, besides the same intended meaning of the symbols of Example 2.1.4 with the addition of a symbol 'diff' that represents the differentiation of a function at a point. Some terms defined over this framework are:*

- $t_1 \equiv \lambda x.add(x, s(zero)) :: nat \to nat$ *represents a function that is the addition of its parameter* x *to* 1*.*
- $t_2 \equiv \lambda x, y.mult(x, y) :: nat \to nat \to nat$ *represents a function that is the multiplication of two natural numbers* x *and* y*.*

- $t_3 \equiv \lambda x.diff\,(\lambda y.y, x) :: nat \to nat$ *represents the differential of the identity function at the point represented as the parameter of the function.*

Terms t_1*,* t_2 *and* t_3 *are total terms, and* ${t_1}' \equiv \lambda x.sum(\bot, s(zero))$ *is a term that verifies* ${t_1}' \sqsubseteq t_1$*. There are more terms lesser than* t_1*, for example, we have:* $\lambda x.\bot \sqsubseteq \lambda x.sum(x, \bot) \sqsubseteq \lambda x.sum(x, s(\bot)) \sqsubseteq \lambda x.sum(x, s(zero))$*. On the other hand,* $\lambda x.sum(x, \bot)$ *and* $\lambda x.sum(\bot, s(zero))$ *cannot be compared with this order because they have completely defined information at different positions.*

The concept of subterm is very similar to the one given for term rewriting systems, and subterms are also identified by a position taken from a set of positions. Here positions are sequences of natural numbers being ϵ the empty sequence; the sequence of n 1's is represented as 1^n, and the concatenation of sequences is represented by the symbol '$\cdot$'. With this notation, we consider terms by using the generic form $\lambda\overline{x_k}.a(\overline{t_n})$ and we use a sequence of 1's to traverse the abstracted variables; in positions that have k 1's the next symbol represents the order of the subterm in the application the position refers to. We define a prefix order relation among positions denoted with the symbol $\preceq$ and defined as $p \preceq q$ if and only if there exists r such that $p \cdot r = q$.

Definition 2.3.3 (Positions and subterms) *Let* $t \equiv \lambda\overline{x_k}.a(\overline{t_n}) \in T(\Sigma_\bot, \mathcal{V})$*:*

- *The set of* ***positions*** *in* t *is denoted as* $Pos(t)$ *and is defined as* $Pos(\lambda\overline{x_k}.a(\overline{t_n})) = \{1^i \mid 0 \leq i \leq k\} \cup \{1^k \cdot j \cdot q \mid 1 \leq j \leq n, q \in Pos(t_j)\}$*.*
- *The* ***subterm of*** t ***at position*** $p \in Pos(t)$ *is denoted as* $t|_p$ *and is defined as follows:*
  - $\lambda x_{i+1}.\cdots.x_k.a(\overline{t_n})$ *if* $p = 1^i$*,* $0 \leq i \leq k$*.*
  - $t_i|_q$ *if* $p = 1^k \cdot i \cdot q$*,* $1 \leq i \leq n$*.*
- *A position* p *is* ***maximal*** *in* t *if* $t|_p$ *is of base type in* $\mathcal{B}$*. The set of maximal positions in a term* t *is denoted as* $MPos(t)$*.*

In this framework it is useful to give a more general notion to the concept of bound variable in a term t, that keeps track of the variables abstracted at any position of the term and not only on the whole term. First we define the ordered sequence of variables abstracted on the path to a position p (shortly, $seq_{\mathcal{BV}}(t, p)$), and from that sequence we define the set of all of them (shortly, $\mathcal{BV}(t, p)$); the set of bound variables of a term is the set that contains the variables abstracted into the path to any position, that is $\mathcal{BV}(t) = \bigcup_{p \in Pos(t)} \mathcal{BV}(t, p)$.

Definition 2.3.4 (Abstracted variables) *Let* $x \in \mathcal{V}$*,* $s, t \in T(\Sigma_\bot, \mathcal{V})$*,* $p, q \in Pos(t)$*:*

- *The* ***sequence of variables abstracted on the path to position*** p *is denoted as* $seq_{\mathcal{BV}}(t,p)$*, and is defined as follows:*

  - $seq_{\mathcal{BV}}(t,\epsilon) = \epsilon$.
  - $seq_{\mathcal{BV}}(\lambda x.s, 1 \cdot q) = x \cdot seq_{\mathcal{BV}}(s,q)$.
  - $seq_{\mathcal{BV}}(h(\overline{t_n}), i \cdot q) = seq_{\mathcal{BV}}(t_i,q)$, *with* $0 < i \leq n$.

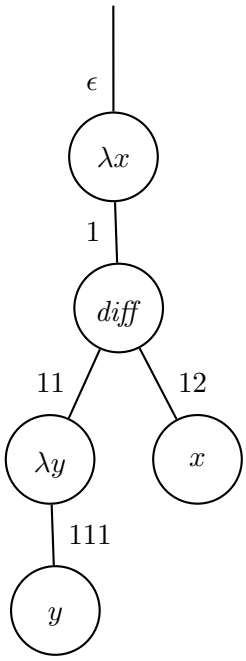
- *The* ***set of variables abstracted on the path to position*** p *is denoted as* $\mathcal{BV}(t,p)$*, and is defined as* $\mathcal{BV}(t,p) = \{x | x \text{ is an element of } seq_{\mathcal{BV}}(t,p)\}$.

The notation $t\restriction_p$ is used to denote the subterm of t at position p with all the bound variable until that position abstracted, that is, $t\restriction_p = \lambda\overline{x_k}.(t|_p)$ where $\overline{x_k} = seq_{\mathcal{BV}}(t,p)$.

Example 2.3.5 (Subterms and positions) *The set of positions for term* $t_3 \equiv \lambda x.diff(\lambda y.y, x)$ *is* $Pos(t_3) = \{\epsilon, 1, 11, 111, 12\}$ *and the subterm corresponding to each of them is:*

$p \in Pos(t_3)$	$t_3\|_p$
ϵ	$\lambda x.diff(\lambda y.y, x)$
1	$diff(\lambda y.y, x)$
11	$\lambda y.y$
111	y
12	x

The evaluation of the subterm corresponding to a given position is similar to the process of traversing the tree related to the term that has its subtrees numbered in the same way that positions are defined, that is a natural and common way of numbering trees:



Finally, we define the *conditional pattern rewriting systems (CPRS)* as higher-order extensions of term rewriting systems. We are going to use this class of higher-order rewriting systems as the basis of our programming framework. The higher-order extension is done by using simply-typed λ-calculus at the syntactic level to have a natural syntax to define higher-order functions in addition to first-order terms of term rewriting systems. Unfortunately, as will be discussed in the next chapter, higher-order unification for generic simply-typed λ-terms is undecidable (indeed, it is undecidable for the second-order case, see [37]), and is a key feature in any goal solving calculus and declarative semantics defined for the language. Because of that, we have to restrict the kind of terms that appear in the programs, considering a subset of simply-typed λ-terms where higher-order unification and pattern matching are both decidable. This subset is the one of the so called higher-order fully-extended *patterns*; patterns are terms that have the limitation that any subterm with a free variable on its head have only bound variables as arguments and all of them are distinct; a pattern is *fully-extended* if there are not additional bound variables in the path to the position of higher-order variables (see [70]). Restricting the set of terms in the language to patterns obviously limits its expressiveness, but it keeps the same computational capacity.

Definition 2.3.6 (Patterns)

- *Let $t \in T(\Sigma_\perp, \mathcal{V})$. Term t is a higher-order* ***pattern*** *if for all $p \in MPos(t)$ such that $hd(t|_p) \in \mathcal{FV}(t)$ and $t|_p = X(\overline{t_n})$ verifies that $t_1 \downarrow_\eta \in \mathcal{BV}(t,p), \ldots, t_n \downarrow_\eta \in \mathcal{BV}(t,p)$ is a sequence of distinct elements.*
- *Let $t \in T(\Sigma_\perp, \mathcal{V})$. Term t is a higher-order* ***fully-extended pattern*** *if is a pattern, and additionally, for all $p \in MPos(t)$ such that $hd(t|_p) \in \mathcal{FV}(t)$ and $t|_p = X(\overline{t_n})$ verifies that $\mathcal{BV}(t,p) \setminus \{t_1 \downarrow_\eta, \ldots, t_n \downarrow_\eta\} = \emptyset$.*

Example 2.3.7 (Patterns) *We consider some examples of patterns, taking into account $f/1 :: nat \to nat$, $F :: nat \to nat \to nat$, and $G :: (nat \to nat) \to nat$:*

- *$\lambda x, y.F(x,y) :: nat \to nat \to nat$ and $\lambda x.f(G(\lambda z.x(z))) :: nat \to nat$ are fully extended linear patterns.*
- *$\lambda x, y, z.f(F(x,y)) :: nat \to nat \to nat \to nat$ is a pattern but is not fully extended because z does not appear among the arguments of the free variable F.*
- *$\lambda x, y.F(zero, y)$ is not a pattern because the first argument of the variable F is not a bound variable, and $\lambda x.G(H(x))$ is not a pattern because $H(x) \downarrow_\eta = H(x)$ is not a bound variable.*

In our higher-order declarative semantic framework, *programs* are a special kind of conditional rewriting systems over fully extended *linear* (i.e., no variable appears twice) patterns, with conditional equations between total terms. In this context, an *equation* is a multiset $\{\!\{s,t\}\!\}$ written $s == t$, where $s, t \in T(\Sigma_\perp, \mathcal{V})$ are terms of the same type.

Definition 2.3.8 (Conditional Pattern Rewriting System)
A ***conditional pattern rewriting system*** *(or CPRS for brevity) is a finite set of conditional rewrite rules of the form $f(\overline{l_n}) \to r \Leftarrow C$, where*

- *$f(\overline{l_n})$ and r are total terms of the same base type in $\mathcal{B}$.*
- *$f(\overline{l_n})$ is a fully extended linear pattern.*
- *C is a (possibly empty) finite sequence of equations between total terms.*

Usually, in a rule of the form $f(\overline{l_n}) \to r \Leftarrow C$, term $f(\overline{l_n})$ is called the *left-hand side* (*lhs* for brevity) of the rule, r the *right-hand side* (*rhs* for brevity) and C the conditional part of the pattern rewrite rule. Each *CPRS* induces a partition of the set of function symbols Σ into the set of symbols that have rules with them in the

head of the left-hand side, called *defined function symbols* (and denoted as Σ_d); and the set of symbols that do not satisfy that condition, called *data constructor symbols* (and denoted as Σ_c). A *CPRS* is said to be *constructor-based* if each conditional pattern rewrite rule $f(\overline{l_n}) \to r \Leftarrow C$ satisfies the condition that $l_1, \ldots, l_n \in T(\Sigma_c, \mathcal{V})$. An alternative and somewhat equivalent way of considering this point is assuming the existence of an universal signature $\Sigma = \Sigma_d \cup \Sigma_c$ that explicitly provides that partition.

Example 2.3.9 (*CPRS*) *We define the CPRS with the rewriting rules for the functions described in Example 2.3.2:*

$$
\begin{array}{lcl}
add(x, zero) & \to & x \\
add(x, s(y)) & \to & s(add(x, y)) \\
 & & \\
mult(x, zero) & \to & zero \\
mult(x, s(y)) & \to & add(x, mult(x, y)) \\
 & & \\
diff\,(\lambda u.F, x) & \to & zero \\
diff\,(\lambda u.u, x) & \to & s(zero) \\
diff\,(\lambda u.add(F(u), G(u)), x) & \to & add(diff\,(\lambda u.F(u), x), diff\,(\lambda u.G(u), x)) \\
diff\,(\lambda u.mult(F(u), G(u)), x) & \to & add(mult(diff\,(\lambda u.F(u), x), G(x)), \\
 & & \quad mult(diff\,(\lambda u.G(u), x), F(x)))
\end{array}
$$

Finally we provide a classification of maximal positions that will be useful in the rest of this work.

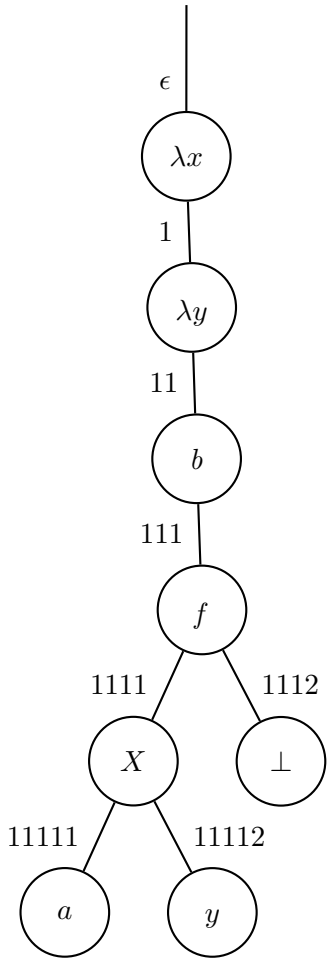
Definition 2.3.10 (Classification of maximal positions) *Let $t \in T(\Sigma_\perp, \mathcal{V})$ and $p \in MPos(t)$:*

- *A maximal position p is **flex** in t if $\forall q \in MPos(t), q \preceq p.\, hd(t|_q) \in \mathcal{FV}(t, q)$.*
- *A maximal position p is **rigid** in t if $\forall q \in MPos(t), q \preceq p. hd(t|_q) \in \mathcal{BV}(t, q) \cup \Sigma$. The set of rigid positions in a term t is denoted as $Pos_r(t)$.*
- *A maximal position p is **safe** in t if $\forall q \in MPos(t), q \preceq p. hd(t|_q) \in \mathcal{BV}(t, q) \cup \Sigma_c$. The set of safe positions in a term t is denoted as $Pos_s(t)$.*

Example 2.3.11 *Let $t \equiv \lambda x, y.b(f(X(a, y), \perp))$, where $f \in \Sigma_d$, $a, b \in \Sigma_c$, a, x, y are of base type τ, $b :: \tau \to \tau$ and $f, X :: \tau \to \tau \to \tau$. We have the following set of positions $Pos(t)$ with their corresponding subterms:*

$p \in Pos(t)$	$t\|_p$
ϵ	$\lambda x, y.b(f(X(a,y),\bot))$
1	$\lambda y.b(f(X(a,y),\bot))$
11	$b(f(X(a,y),\bot))$
111	$f(X(a,y),\bot)$
1111	$X(a,y)$
1112	$\bot$
11111	a
11112	y

The following tree represents the positions of this example:



The sets of positions defined in Definition 2.3.10 are the following:

- $MPos(t) = \{11, 111, 1111, 1112, 11111, 11112\}$.
- $Pos_r(t) = \{11, 111\}$.
- $Pos_s(t) = \{11\}$.
- No position in t is flex.

Chapter 3

A Higher-Order Rewriting Logic with λ-Abstractions

In this chapter we present a rewriting logic for the higher-order programming language with λ-abstractions introduced in the previous Chapter 2. This logic defines the reduction relation among terms for a given *CPRS* in a natural way that is also useful to reason about programs and has been adapted from [23] and [24]. In the first section of this chapter we discuss the problem of higher-order unification that motivates the apparently arbitrary election of patterns defined in Chapter 2; our proposal of an unification algorithm for higher-order patterns can be found in Appendix A, with a proof of its main properties of soundness and completeness. In the second section we describe the rewriting logic *GHRC* and its associated proof calculus. In the last section of the chapter, we present some properties about the proof calculus associated to this logic that are essential to prove basic results about the declarative semantics of the higher-order functional logic programming language in Chapter 4.

3.1 Higher-Order Unification

In this section we discuss the problem of unification of higher-order terms in the presence of λ-abstractions and higher-order variables motivating the necessity of an specialized algorithm, that is presented in Appendix A. Also we give some auxiliary notions needed to define the unification algorithm and the reduction relation in Section 3.2.

Unification and pattern matching are key features of any rewriting framework that has an operational semantics based on narrowing and rewriting [8]. It is needed to give suitable notions of reduction and operational semantics, and it is required to be efficient in any implementation based on the framework. In the first-order case

of term rewriting systems described in Section 2.1, the unification algorithm referenced there has linear complexity with respect to the structure of the terms unified and returns an unique most general unifier (see [66]). The general higher-order case presents some differences that are discussed in this section. The ideas presented here and some of the examples are taken from [28, 81].

As precisely defined for the first-order case in Section 2.1.2, the unification problem can be summarized as follows: starting with two terms $s, t \in T(\Sigma_\perp, \mathcal{V})$, solving an unification problem $s \approx t$ consists on finding a substitution σ such that $\sigma(s)$ and $\sigma(t)$ are syntactically the same; substitution σ is the *most general unifier* (see Definition 2.1.12) of s and t if every unifier τ of s and t holds that $\sigma \leq \tau$. In the first-order case, the algorithm presented in [66] looks for the unifier essentially by traversing simultaneously the syntactic structure of s and t, and when it finds a position where one term has a variable and the other has a different (possibly non-variable) term, it binds the variable to the other term. The algorithm also detects terms that cannot be unified; there are two possible reasons to this case detected by the algorithm: one, if there is a position where the head symbols in both terms are not variables and are different (for example, position 1 in $f(g(x))$ and $f(h(x))$); the other happens when a variable is tried to be unified with a term that contains it (for example, $x \approx f(x)$ cannot be unified) and the checking of this condition is called *occurs check*.

In the higher-order case we are looking for a substitution σ that solves an unification problem $s \approx t$ such that $\sigma(s)\updownarrow_\beta^\eta$ and $\sigma(t)\updownarrow_\beta^\eta$ are syntactically the same. In this case, considering $\beta\eta$-reduction as an implicit operation, the treatment of free and bound variables causes some particular problems that must be treated carefully. With respect to bound variables, it is needed to be careful with the arguments that terms assigned to higher-order variables can have; for example, considering a signature $\Sigma = \{f/1 :: nat \to nat, g/1 :: nat \to nat\}$, the unification problem $\lambda x.f(F(x)) \approx \lambda x.f(g(x))$ has the solution $\sigma = \{F \to \lambda y.g(y)\}$, but $\lambda x.F \approx \lambda x.f(g(x))$ has no solution because a term bound to F cannot depend on x in $\lambda x.F$. The unification algorithm has to consider the arguments that a higher-order variable might have when building bindings for them.

The presence of higher-order free variables also causes some particular situations that are not present in the first-order case. For example, the unification problem $a \approx F(a)$, that has no solution in the first-order case, has a couple of incomparable solutions in the higher-order case: $\{F \mapsto \lambda x.x\}$ and $\{F \mapsto \lambda x.a\}$. It can be the case that there are even infinite incomparable unifiers for an unification problem; for example, the problem $F(f(a)) \approx f(F(a))$ has as solution each substitution $\{F \mapsto \lambda x.f^n(x)\}$, for every $n \geq 0$. This means that it is impossible to find a most general unifier as in the first-order case because there might be incomparable subs-

titutions that serve as unifiers; because of this, the concept of most general unifier is generalized to a set of unifiers that is *minimal* in a way that is precisely defined in Appendix A.

The undecidability of higher-order unification was first shown in [63] and even in the second-order case in [37]. *Dale Miller* discovered in 1991 a class of λ-terms with decidable higher-order unification and in the case that more than one unifier exists one of them is a most general unifier (see [70]). This class of λ-terms is the one of the so-called higher-order patterns that were defined in Definition 2.3.6. These patterns have distinct bound variables as arguments of free variables; that makes higher-order unification an extension of first-order unification and preserve some of the good properties of it, for instance, its decidability and linear complexity [83]. That is the reason because higher-order patterns are used at the syntactic level in our framework and the ideas behind the somewhat sophisticated unification algorithm that is presented in Appendix A.

An auxiliary notion we need to introduce is the concept of *lifter*. A lifter is an auxiliary operation that deals with a problem that arises with bound variables when considering subterms of a given term; that is, considering that terms in the framework are in long $\beta\eta$-normal form, when considering a subterm of a given term some bindings of variables may be lost and a variable that was bound in the original term can be free when considering a particular subterm. Also, when trying to unify two terms it may be needed that both of them have the same variables as arguments of free variables so unification can be performed. To solve these two kind of problems, we define the $\overline{x_k}$-*lifter* of a term t with respect to a set of variables V that adds to every free variable in t the variables $\overline{x_k}$ given as arguments, and also adds the abstracted variables in the path to the position of the subterm.

Definition 3.1.1 (Lifter) *Let $t \in T(\Sigma_\perp, \mathcal{V})$:*

- *The $\overline{x_k}$-**lifter of** t **with respect to** $\mathcal{V}$ is a term denoted as $t^{\uparrow \overline{x_k} \restriction \mathcal{V}}$, defined inductively as follows:*
  - *If $t \equiv \lambda \overline{y_l}.\pi$ then $(\lambda \overline{y_l}.\pi)^{\uparrow \overline{x_k} \restriction \mathcal{V}} = \lambda \overline{y_l}.(\pi^{\uparrow \overline{y_l}, \overline{x_k} \restriction \mathcal{V}})$.*
  - *If $t \equiv a(\overline{t_n})$ with $a \notin \mathcal{V}$ then $(a(\overline{t_n}))^{\uparrow \overline{x_k} \restriction \mathcal{V}} = a(\overline{t_n^{\uparrow \overline{x_k} \restriction \mathcal{V}}})$.*
  - *If $t \equiv X(\overline{t_n})$ with $X \in \mathcal{V}$ then $(X(\overline{t_n}))^{\uparrow \overline{x_k} \restriction \mathcal{V}} = X(\overline{x_k}, \overline{t_n^{\uparrow \overline{x_k} \restriction \mathcal{V}}})$.*
- *The $\overline{x_k}$-**lifter** of t is a term denoted as $t^{\uparrow \overline{x_k}}$ and is defined as $t^{\uparrow \overline{x_k}} = t^{\uparrow \overline{x_k} \restriction \mathcal{FV}(t)}$.*
- *The notation $t^{\updownarrow \overline{x_k}}$ denotes the term $\lambda \overline{x_k}.(t^{\uparrow \overline{x_k}})$.*
- *If $C \equiv \overline{s_n == t_n}$ is a sequence of equations then $C^{\updownarrow \overline{x_k}}$ denotes the sequence $\overline{s_n^{\updownarrow \overline{x_k}} == t_n^{\updownarrow \overline{x_k}}}$.*

Example 3.1.2 (Lifter) *In this example we evaluate the lifter for some term defined from the signature of Example 2.3.9:*

1. $(\lambda x, y.add(F(x,y), \lambda z.mult(G(y))))^{\uparrow v\restriction_{\{F,G\}}} =$
   $\lambda x, y.(add(F(x,y), \lambda z.mult(G(y))))^{\uparrow x,y,v\restriction_{\{F,G\}}} =$
   $\lambda x, y.add((F(x,y))^{\uparrow x,y,v\restriction_{\{F,G\}}}, (\lambda z.mult(G(y)))^{\uparrow x,y,v\restriction_{\{F,G\}}}) =$
   $\lambda x, y.add((F(x,y,v)), (\lambda z.mult(G(y)))^{\uparrow x,y,v\restriction_{\{F,G\}}}) =$
   $\lambda x, y.add((F(x,y,v)), \lambda z.(mult(G(y)))^{\uparrow z,x,y,v\restriction_{\{F,G\}}}) =$
   $\lambda x, y.add((F(x,y,v)), \lambda z.(mult(G(y)^{\uparrow z,x,y,v\restriction_{\{F,G\}}})) =$
   $\lambda x, y.add((F(x,y,v)), \lambda z.(mult(G(z,x,v,y))))$

2. $(add(F,G))^{\updownarrow x,y} =$
   $\lambda x, y.(add(F,G))^{\uparrow x,y\restriction_{\{F,G\}}} =$
   $\lambda x, y.add(F^{\uparrow x,y\restriction_{\{F,G\}}}, G^{\uparrow x,y\restriction_{\{F,G\}}}) =$
   $\lambda x, y.add(F(x,y), G^{\uparrow x,y\restriction_{\{F,G\}}}) =$
   $\lambda x, y.add(F(x,y), G(x,y))$

In order to define the reduction relation underlying the proof calculus of our higher-order rewriting logic *GHRC*, we are interested in terms that will play a similar role to normal forms in term rewriting systems, that is, terms that cannot be reduced with respect to a program. This is an extension of the concept of *c-term* usual in standard first-order functional logic frameworks (see, e.g., [22]): in that general context, a c-term is a term that contains only data constructor and variable symbols; intuitively, those terms cannot be reduced since rules can only be applied to positions with a function symbol on it. In our higher-order framework, we have to deal with higher-order variables, so we extend the concept of c-term to the concept of *value*; values do not have subterms that unify with the left-hand side of any program rule of a given *CPRS*, so no rewrite steps can be performed from them.

Definition 3.1.3 (Values) *Let $t \in T(\Sigma_\perp, \mathcal{V})$ and $\mathcal{R}$ a CPRS:*

- *A term t is a **value** if and only if for all $p \in MPos(t)$ and $(\pi \to r \Leftarrow C) \in \mathcal{R}$ such that $\mathcal{FV}(t) \cap \mathcal{FV}(\pi) = \emptyset$ there is no solution to $t\restriction_p \approx \pi^{\updownarrow seq_{\mathcal{BV}}(t,p)}$. The set of all values is denoted as $Val(\Sigma_\perp, \mathcal{V})$.*
- *A term t is a **total value** if it is a value and it is a total term. The set of all total values is denoted as $Val(\Sigma, \mathcal{V})$.*
- *A **value substitution** is a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $t_1, \ldots, t_n$ are values. The set of all value substitutions is denoted as $VSubst(\Sigma_\perp, \mathcal{V})$.*

In contrast to the concept of value substitution, we denote as $Subst(\Sigma_\perp, \mathcal{V})$ the set of substitutions that map variables to any term in $T(\Sigma_\perp, \mathcal{V})$. We will also denote as $[\mathcal{R}]_\perp$ the set of all instances of a rule affected by a value substitution, that is:

$$[\mathcal{R}]_\perp = \{(l \to r \Leftarrow C)\,\theta \mid (l \to r \Leftarrow C) \in \mathcal{R} \text{ and } \theta \in \mathit{VSubst}(\Sigma_\perp, \mathcal{V})\}$$

Example 3.1.4 (Values) *The following terms are values with respect to the CPRS of Example 2.3.9:*

1. $s(s(zero))$.
2. $\lambda x.s(s(x))$.

The following terms are not values with respect to the same CPRS:

3. $\lambda x, y.add(s(x), s(y))$.
4. $\lambda x, y.F(s(x), s(y))$.

3.2 The Higher-Order Proof Calculus *GHRC*

In this section we present the higher-order logical framework called *GHRC* that infers *reduction* and *equality* statements by means of a proof calculus, also called *GHRC* (*Goal-oriented Higher-order Rewriting Calculus*); it is also a suitable calculus to reason about programs in fields as *debugging* and *verification* [24, 26]. It also clarifies the reduction relation that we are going to introduce by means of the proof calculus of the logic adapting the ideas presented in Section 2.1.3.

The *GHRC* logic is an extension to deal with higher-order conditional rewrite rules of the first-order *constructor-based conditional rewriting logic* (*CRWL* for short) schema described in [38]. In contrast to Meseguer's logic (see [68]), which aims at modeling change caused by concurrent actions at a very high abstraction level, our rewriting logic intends to model the evaluation of λ-terms in a constructor-based language involving non-strict lazy functions with *call-time choice*. As in [38], we do not impose non-ambiguity conditions; this means that non-deterministic functions are allowed.

First of all we present the kind of statements we are going to infer from the proof calculus later in this section. Given a *CPRS* $\mathcal{R}$ we can derive statements of the two following kinds:

- *Reduction statements* $s \twoheadrightarrow t$, where $s, t \in T(\Sigma_\perp, \mathcal{V})$ and s and t have the same type from $\mathcal{B}$.
- *Equality statements* $s == t$, where $s, t \in T(\Sigma, \mathcal{V})$ and s and t have the same type from $\mathcal{B}$.

Reduction statements $s \twoheadrightarrow t$ represent reductions of terms, and equality statements $s == t$ represent program or goal conditions by means of two reduction statements $s \twoheadrightarrow u$ and $t \twoheadrightarrow u$ to a common total value $u \in \mathit{Val}(\Sigma, \mathcal{V})$. Rules that define the relation induced by these statements are given by means of the so-called *GHRC* proof calculus.

The *GHRC* proof calculus is defined in Figure 3.1 by means of several inference rules; rules for reduction statements follow the idea of *approximation*: we can consider elements of Σ_d as information that has not been evaluated so it is unknown; in that case we can consider that the computed information of a term consists on the term replacing any subterm with a function symbol at the head by the constant $\bot$ corresponding to its type. In that case, the proof calculus refines the term by means of the application of program rules to function symbols replacing active subterms by more defined comparable terms considering the approximation ordering $\sqsubseteq$ between terms defined in Section 2.3. It is also possible by means of the calculus to substitute any information by a constant $\bot$, to represent information that is not needed to be computed in order to proceed with the proof derivation.

There is only one unique rule for equality statements $s == t$ that precisely defines equality conditions in this framework as joinability of s and t to a common total value u. This follows the idea of specifying joinability as a generalization of strict equality, where total values in our higher-order framework play the same role as total constructor terms in a first-order framework (see [38]). This is easily expressed by the following formal rule:

$$\frac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t} \text{ if } u \in \mathit{Val}(\Sigma, \mathcal{V}).$$

Now we present the rules for reduction statements. These rules encapsulate the desired specification of the language, that is, non-strict lazy non-deterministic functions with *call-time choice* [38].

Any pattern can be reduced to a corresponding bottom constant, actually reducing it to the minimum comparable element with respect to the approximation order '$\sqsubseteq$' between terms. This represents computations that do not terminate or are not needed to be performed in order to find solutions to a goal in the operational semantics, that is, a non strict argument of a function can be substituted by a constant $\bot$ because it is not demanded when applying a rule; at this level, it is manifested by the fact that any term can be reduced to a corresponding constant $\bot$ in one step of the calculus. Because of that, the *bottom rule* **B** is very simple:

$$\lambda\overline{x_k}.\, \pi \twoheadrightarrow \lambda\overline{x_k}.\, \bot$$

The bottom rule is one of the base rules of the proof calculus; it allows to finish derivations by substituting any pattern by a bottom constant; the other way of finishing derivations is leaving the current computed term as it is. Depending on the objective of the proof derivation, the current proof state can be an acceptable result; for example, when computing a proof for an equality statement, it is required that the common term obtained by reducing two terms to be a total value, so if the current term in a proof derivation for one of them is a total value there is no need to reduce it anymore and the proof can be finished at the current state. The *reflexivity rule* **RF** that expresses this behavior is also simple:

$$s \twoheadrightarrow s$$

The first inductive rule of the calculus we present is the *monotonicity rule* **MN**, defined for all the symbols of $\Sigma \cup \mathcal{V}$. It computes a more defined term by refining the arguments of a function or variable symbol. This rule allows to reduce the arguments of a symbol as desired applying other rules of the calculus:

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow \lambda\overline{x_k}.\, t_1 \;\ldots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow \lambda\overline{x_k}.\, t_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\, a(\overline{t_n})}$$

The other inductive rule of the calculus is the one that defines how terms are reduced by means of applying program rules to them. This rule is called *outermost reduction* **OR** because the program rule is applied to the outermost symbol of a term. To do that, the symbol at the head of the program rule must be the same that the symbol at the head of the term, the parameters of the term must be reduced to the formal parameters of the program rule that is applied and the (possible) conditions of the rule must be fulfilled; if that is the case, then the rule can be applied and the term can be rewritten to the right-hand side of the rule instantiated by the substitution inferred in the process. The rule that reflects this behavior is the following:

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}}\theta \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}}\theta \quad C^{\updownarrow\overline{x_k}}\theta \quad r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}$$

if $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ and $\theta \in \mathit{VSubst}(\Sigma_\perp, \mathcal{V})$, for any $u \neq \lambda\overline{x_k}.\, \perp$.

It is interesting to notice the use of the lifter $\updownarrow \overline{x_k}$ in this rule; program rules are generic entities that have to deal with many different terms, and any of them can have any number of different abstracted variables; the lifter ensures that the parameters in the program rule have the same bound variables that the term that is being reduced, opening the possibility of applying the rule with independence of the variables abstracted until the subterm at the position where the rule is applied. It is also interesting to notice that arguments in the term being reduced have to be reduced to instances of the corresponding parameters of the program rule by

means of a substitution $\theta \in \mathit{VSubst}(\Sigma_\perp, \mathcal{V})$; this helps to ensure the call time-choice parameter passing since this substitution is common to all the parameters $\overline{l_n}$ and to the resulting term u.

It is usual to split the **OR** rule to make explicit in the calculus the function that is applied. This have applications in verification and debugging (see e.g., [14, 15, 24]). In that case the rule **OR** is replaced by:

$$\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow l_1^{\downarrow\overline{x_k}}\theta \;\ldots\; \lambda\overline{x_k}.s_n \twoheadrightarrow l_n^{\downarrow\overline{x_k}}\theta \qquad \dfrac{C^{\downarrow\overline{x_k}}\theta \qquad r^{\downarrow\overline{x_k}}\theta \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \twoheadrightarrow u}}}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u}$$

Now **OR** rule is separated in one rule to perform the parameter passing, named *argument reduction* and denoted **AR**, and another for the rule application that includes checking the equations on its conditional part (if there are any), named *rule application* and denoted **RA**. The function call enclosed in a box is called a *basic fact* and is used for verification and debugging purposes [14, 15, 16]. The rules are now the following, under the same premises of rule **OR**:

AR:

$$\frac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\downarrow x_k} \;\cdots\; \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\downarrow x_k} \qquad \boxed{\lambda\overline{x_k}.\, f(\overline{l_n^{\downarrow x_k}}) \twoheadrightarrow u}}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}$$

RA:

$$\frac{C^{\downarrow\overline{x_k}} \qquad r^{\downarrow\overline{x_k}} \twoheadrightarrow u}{\boxed{\lambda\overline{x_k}.\, f(\overline{l_n}) \twoheadrightarrow u}}$$

From the calculus of Figure 3.1, it is possible to build *proof trees* for reduction and equality statements. We denote the *set of proof trees* for a reduction or equality statement φ as $\mathcal{PT}(\varphi)$ and the set of proof trees ending with the application of a rule $\mathbf{R}$ of the calculus as $\mathcal{PT}_{\mathbf{R}}(\varphi)$. We say that the statement φ is provable from the calculus and a *CPRS* $\mathcal{R}$ if there exists a proof tree for it and we denote it as $\mathcal{R} \vdash \varphi$; if there exists a proof tree ending with rule $\mathbf{R}$ we denote it as $\mathcal{R} \vdash_{\mathbf{R}} \varphi$. We denote the negation of that relations respectively as $\mathcal{R} \nvdash \varphi$ and $\mathcal{R} \nvdash_{\mathbf{R}} \varphi$.

In order to complete the presentation of our higher-order framework in a declarative programming setting, we give a definition for the class of *goals* from a given

B	**Bottom**	$\lambda\overline{x_k}.\,\pi \twoheadrightarrow \lambda\overline{x_k}.\,\bot$
RF	**ReFlexivity**	$s \twoheadrightarrow s$
MN	**MoNotonicity**	$\dfrac{\lambda\overline{x_k}.\,s_1 \twoheadrightarrow \lambda\overline{x_k}.\,t_1 \;\ldots\; \lambda\overline{x_k}.\,s_n \twoheadrightarrow \lambda\overline{x_k}.\,t_n}{\lambda\overline{x_k}.\,a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\,a(\overline{t_n})}$
OR	**Outermost Reduction**	$\dfrac{\lambda\overline{x_k}.\,s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}}\theta \;\cdots\; \lambda\overline{x_k}.\,s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}}\theta \quad C^{\updownarrow\overline{x_k}}\theta \quad r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.\,f(\overline{s_n}) \twoheadrightarrow u}$ if $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$, $u \neq \lambda\overline{x_k}.\,\bot$, $\theta \in \mathit{VSubst}(\Sigma_\bot, \mathcal{V})$.
J	**Join**	$\dfrac{s \twoheadrightarrow u \qquad t \twoheadrightarrow u}{s == t}$ if $u \in \mathit{Val}(\Sigma, \mathcal{V})$.

Figure 3.1: The *GHRC* proof calculus.

CPRS $\mathcal{R}$ we want to evaluate, and the kind of *solutions* of a goal we want to compute by using an appropriate operational semantics based on *higher-order narrowing* (see e.g., [23, 47]). Goals are simply defined as sets of equality statements, that must be fulfilled in order to solve the goal. Solutions are common substitutions to all the equality statements in the goal that allow to reduce them to a common total value by means of the *GHRC* proof calculus.

Definition 3.2.1 (Goals and Solutions)

- *Let* $\overline{s_n}, \overline{t_n} \in T(\Sigma_\bot, \mathcal{V})$ *such that for all* $i \in \{1, \ldots, n\}$ *both* s_i *and* t_i *has the same type from* $\mathcal{B}$. *A* ***goal*** G *for a given CPRS* $\mathcal{R}$ *is a set* $\{\overline{s_n == t_n}\}$ *of equations. Equations are symmetric:* $s == t \equiv t == s$.

- $\gamma \in \mathit{Subst}(\Sigma_\bot, \mathcal{V})$ *is a* ***solution*** *of a goal* $G \equiv \{\overline{s_n == t_n}\}$ *if it holds that* $\gamma\restriction_{\mathcal{FV}(G)} \in \mathit{VSubst}(\Sigma_\bot, \mathcal{V})$, *and for each equation* $s_i == t_i$ *in* G *there exists a proof tree* $\mathcal{P}_i \in \mathcal{PT}(s_i\gamma == t_i\gamma)$. *The proof tree* $\mathcal{P}_i$ *is called a* ***witness*** *that* γ *is a solution of* $s_i == t_i$.

- *We write $Soln(G)$ for the* ***set of solutions*** *of a goal G, and $Wtn_\gamma(G)$ for the* ***set of witnesses*** *that γ is a solution of G.*

Example 3.2.2 (*GHRC* proof trees) *In this example we use the GHRC proof calculus to present a proof tree of a reduction statement with respect to the CPRS defined in Example 2.3.9. These kind of examples are quite laborious for proving simple goals, have a lot of redundant parts and have correct guesses of the rule to apply that would be very difficult to do by an automatic method. Because of these reasons, this logic is not suitable to be directly the basis of an operational semantic. As we have mentioned, a suitable operational semantic for this framework that affords the problems described before is based on needed narrowing with definitional trees and is presented in [23].*

We prove that $(x)'(0) * 2 = 2$ by building a proof tree for the reduction statement $mult(diff(\lambda x.x,\ zero), s(s(zero))) \twoheadrightarrow s(s(zero))$:

OR $mult(diff(\lambda x.x, zero), s(s(zero))) \twoheadrightarrow s(s(zero))$
- **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
  - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
  - **RF** $zero \twoheadrightarrow zero$
  - **RF** $s(zero) \twoheadrightarrow s(zero)$
- **RF** $s(s(zero)) \twoheadrightarrow s(s(zero))$
- **OR** $add(diff(\lambda x.x, zero), mult(diff(\lambda x.x, zero), s(zero))) \twoheadrightarrow s(s(zero))$
  - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
    - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
    - **RF** $zero \twoheadrightarrow zero$
    - **RF** $s(zero) \twoheadrightarrow s(zero)$
  - **OR** $mult(diff(\lambda x.x, zero), s(zero)) \twoheadrightarrow s(zero)$
    - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
      - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
      - **RF** $zero \twoheadrightarrow zero$
      - **RF** $s(zero) \twoheadrightarrow s(zero)$
    - **RF** $s(zero) \twoheadrightarrow s(zero)$
    - **OR** $add(diff(\lambda x.x, zero), mult(diff(\lambda x.x, zero), zero)) \twoheadrightarrow s(zero)$
      - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
        - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
        - **RF** $zero \twoheadrightarrow zero$
        - **RF** $s(zero) \twoheadrightarrow s(zero)$
      - **OR** $mult(diff(\lambda x.x, zero), zero) \twoheadrightarrow zero$
        - **OR** $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
          - **RF** $\lambda x.x \twoheadrightarrow \lambda x.x$
          - **RF** $zero \twoheadrightarrow zero$
          - **RF** $s(zero) \twoheadrightarrow s(zero)$
        - **RF** $zero \twoheadrightarrow zero$
        - **RF** $zero \twoheadrightarrow zero$

RF $s(zero) \twoheadrightarrow s(zero)$
MN $s(add(diff(\lambda x.x, zero), zero)) \twoheadrightarrow s(s(zero))$
OR $add(diff(\lambda x.x, zero), zero) \twoheadrightarrow s(zero)$
OR $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$
RF $zero \twoheadrightarrow zero$
OR $diff(\lambda x.x, zero) \twoheadrightarrow s(zero)$
RF $\lambda x.x \twoheadrightarrow \lambda x.x$
RF $zero \twoheadrightarrow zero$
RF $s(zero) \twoheadrightarrow s(zero)$

3.3 Properties of the Higher-Order Logic *GHRC*

In this section we prove some lemmas describing interesting properties about the *GHRC* logic that will be useful in the next chapter when proving characterization theorems about the declarative semantics of the higher-order programming language and are compiled from [24].

The first result we prove is the so-called *approximation property*. Values in a *CPRS* as defined in Definition 3.1.3 are terms that cannot be reduced applying function rules with respect to it; this lemma states that if a partial value s can be reduced to a value t, then t must be another partial value that contains at most the same information as s, and in the reduction there cannot be applied function rules from the program. Also if s is a total value then t must be a total value too, that is obvious since total values are maximal values with respect to the approximation ordering '$\sqsubseteq$' (because they do not contain $\bot$ symbols). This lemma characterizes the way values are intended to behave with respect to a program: they are terms that have computed the more possible information with respect to the function rules of the program and can only be reduced to less defined terms.

Lemma 3.3.1 (Approximation property) *Let $s \in Val(\Sigma_\bot, \mathcal{V})$. If $\mathcal{R} \vdash s \twoheadrightarrow t$ then $t \in Val(\Sigma_\bot, \mathcal{V})$, $s \sqsupseteq t$, and $\mathcal{R} \nvdash_{\mathbf{OR}} s \twoheadrightarrow t$. Moreover, if $t \in Val(\Sigma, \mathcal{V})$ then $s \equiv t$.*

Proof By structural induction on the proof tree $\mathcal{P}$ for the statement of the form $s \twoheadrightarrow t$ with $s \in Val(\Sigma_\bot, \mathcal{V})$.

- If $\mathcal{P} \in \mathcal{PT}_{\mathbf{B}}(s \twoheadrightarrow t)$ then $t \equiv \bot$, and trivially $\bot \in Val(\Sigma_\bot, \mathcal{V})$ and $s \sqsupseteq \bot$.
- If $\mathcal{P} \in \mathcal{PT}_{\mathbf{RF}}(s \twoheadrightarrow t)$ then $t \equiv s$, and by hypothesis $s \in Val(\Sigma_\bot, \mathcal{V})$ and trivially $s \sqsupseteq s$.

- If $\mathcal{P} \equiv \frac{\mathcal{P}_1 \ldots \mathcal{P}_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\, a(\overline{t_n})}(\mathbf{MN})$ with $\mathcal{P}_i \equiv \mathcal{PT}(\lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i)$ then for induction hypothesis we know that $\lambda\overline{x_k}.\, t_i \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$ and $\lambda\overline{x_k}.\, t_i \sqsubseteq \lambda\overline{x_k}.\, s_i$ for all $i \in \{1, \ldots, n\}$. These relations imply $t \equiv \lambda\overline{x_k}.\, a(\overline{t_n}) \sqsubseteq \lambda\overline{x_k}.\, a(\overline{s_n}) \equiv s$ because $\frac{s_1 \sqsupseteq t_1 \ \cdots \ s_n \sqsupseteq t_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \sqsupseteq \lambda\overline{x_k}.\, a(\overline{t_n})}$ and finally $s \sqsupseteq t$.

  To prove that $t \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$, we know by induction hypothesis that $\lambda\overline{x_k}.\, t_i \in \mathit{Val}(\mathcal{F}_\perp, \mathcal{V})$ for all $i \in \{1, \ldots, n\}$ and we show that t does not match $\pi^{\downarrow\overline{x_k}}$ whenever π is the left-hand side of a rewrite rule. Suppose by contrary that this is not the case. Since $\pi^{\downarrow\overline{x_k}}$ is a linear pattern and $t \sqsubseteq s$, s matches $\pi^{\downarrow\overline{x_k}}$ too. Since this contradicts $s \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$, we conclude $t \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$.

- We show that $\mathcal{P} \notin \mathcal{PT}_{\mathbf{OR}}(s \twoheadrightarrow t)$. Assume by contrary that $\mathcal{P} \in \mathcal{PT}_{\mathbf{OR}}(s \twoheadrightarrow t)$. Then, $s \equiv \lambda\overline{x_k}.\, f(\overline{s_n})$ and there exists $(l \to r \Leftarrow C) \in \mathcal{R}$ and n subtrees $\mathcal{P}_i \in \mathcal{PT}(\lambda\overline{x_k}.\, s_i \twoheadrightarrow l_i^{\downarrow\overline{x_k}}\theta)$ of $\mathcal{P}$ for $1 \leq i \leq n$. By induction hypothesis for $\mathcal{P}_i$ we obtain $\lambda\overline{x_k}.\, l_i^{\downarrow\overline{x_k}}\theta \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$ and $\lambda\overline{x_k}.\, l_i^{\downarrow\overline{x_k}}\theta \sqsubseteq \lambda\overline{x_k}.\, s_i$. Then $f(\overline{l_n})^{\downarrow\overline{x_k}}\theta \sqsubseteq \lambda\overline{x_k}.\, f(\overline{s_n})$. Since $f(\overline{l_n})^{\downarrow\overline{x_k}}$ is a fully-extended linear pattern, this implies the existence of γ such that $f(\overline{l_n})^{\downarrow\overline{x_k}}\gamma = \lambda\overline{x_k}.\, f(\overline{s_n})$. Thus $\lambda\overline{x_k}.\, f(\overline{s_n}) \notin \mathit{Val}(\Sigma_\perp, \mathcal{V})$, that is a contradiction. Finally, from the definition of the approximation ordering, $s, t \in \mathit{Val}(\Sigma, \mathcal{V})$, and $t \sqsubseteq s$ results $s \equiv t$.

□

The second result is the *transitivity property* of the reduction relation with respect to values, and it is an easy consequence of Lemma 3.3.1.

Lemma 3.3.2 (Transitivity property) *If $s, t, u \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$, $\mathcal{R} \vdash s \twoheadrightarrow t$ and $\mathcal{R} \vdash t \twoheadrightarrow u$ then $\mathcal{R} \vdash s \twoheadrightarrow u$.*

Proof Since $s \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$ then $s \sqsupseteq t$ because $\mathcal{R} \vdash s \twoheadrightarrow t$ and $t \sqsupseteq u$ because $\mathcal{R} \vdash t \twoheadrightarrow u$ from Lemma 3.3.1. From the transitivity of '$\sqsupseteq$' we know then that $s \sqsupseteq u$ and because $s, u \in \mathit{Val}(\Sigma_\perp, \mathcal{V})$ then $\mathcal{R} \vdash s \twoheadrightarrow u$ reasoning by induction.

□

The next result we present is a *monotonicity property* on safe positions (correspondent to subterms with a bound variable or a constructor symbol at its head) that states that if $t \twoheadrightarrow u$ with u a total value then any subterm at a safe position in t can be reduced to the subterm at the same position in u. Formally:

Lemma 3.3.3 (Monotonicity property) *If $\mathcal{R} \vdash t \twoheadrightarrow u$, $p \in \mathit{Pos}_s(t)$, and $u \in \mathit{Val}(\Sigma, \mathcal{V})$, then $p \in \mathit{Pos}(u)$ and $\mathcal{R} \vdash t|_p \twoheadrightarrow u|_p$.*

Proof We prove this lemma by induction on the length of p. By definition of *GHRC*, $\mathcal{R} \vdash_{\mathbf{L}} t \twoheadrightarrow u$, where $\mathbf{L} \in \{\mathbf{B}, \mathbf{OR}, \mathbf{RF}, \mathbf{MN}\}$.
Since p is a safe position and u is a total value then $\mathbf{L} \notin \{\mathbf{B}, \mathbf{OR}\}$.
If $\mathbf{L} = \mathbf{RF}$ then $t = u$ and Lemma 3.3.3 holds trivially.
Otherwise, $t = \lambda\overline{x_k}.a(\overline{t_n})$, $u = \lambda\overline{x_k}.a(\overline{u_n})$, and $\mathcal{R} \vdash \lambda\overline{x_k}.t_i \twoheadrightarrow \lambda\overline{x_k}.u_i$ for $i = 1, \ldots, n$:

- If $p = 1^k$ then Lemma 3.3.3 holds trivially.
- Otherwise $p = 1^k \cdot m \cdot q$. Let $p' = 1^k \cdot q$, $t' = \lambda\overline{x_k}.t_m$, and $u' = \lambda\overline{x_k}.u_m$. Then p' is shorter than p, $\mathcal{R} \vdash t' \twoheadrightarrow u'$, $p' \in Pos_s(t')$, and $u' \in Val(\Sigma, \mathcal{V})$. By induction hypothesis for p' we obtain $p' \in Pos_s(u')$, and $\mathcal{R} \vdash t'|_{p'} \twoheadrightarrow u'|_{p'}$. Since $t'|_p = t|_p$ and $u'|_{p'} = u|_p$, we have $\mathcal{R} \vdash t|_p \twoheadrightarrow u|_p$. Also, $p' \in Pos_s(u')$, and $a \in \Sigma_c \cup \{\overline{x_k}\}$ yields $p \in Pos_s(u)$ a safe position.

□

Now we present the following decomposition property of proofs:

Lemma 3.3.4 (Splitting property) *Let consider $s = \lambda\overline{x_i}.\pi'$, $p \in MPos(s)$, $\overline{x_k} = seq_{\mathcal{BV}}(s, p)$, and $\mathcal{P} \in \mathcal{PT}(s|_p \twoheadrightarrow \lambda\overline{x_k}.\pi)$. There exist $\mathcal{P}' \in \mathcal{PT}(s \twoheadrightarrow s[\pi]_p)$ with $|\mathcal{P}'|_{\mathbf{OR}} = |\mathcal{P}|_{\mathbf{OR}}$, where $|\mathcal{P}|_{\mathbf{OR}}$ is the number of applications of* **OR** *in the proof tree $\mathcal{P}$. Moreover, if $p > 1^i$ then $\mathcal{P}' \in \mathcal{PT}_{\mathbf{MN}}(s \twoheadrightarrow s[\pi]_p)$.*

Proof By induction on the length of p. If $p = 1^i$ then we are done because we can choose $\mathcal{P}' = \mathcal{P}$. Otherwise $s = \lambda\overline{x_i}.a(\overline{s_n})$ and $p = 1^i \cdot m \cdot q$ for some $m \in \{1, \ldots, n\}$ such that $1^i \cdot q \in Pos(\lambda\overline{x_i}.s_m)$. Let $p' = 1^i \cdot q$ and $s' = \lambda\overline{x_i}.s_m$. Then $s'|_{p'} = s|_p$, $p' \preceq p$, and we can apply the induction hypothesis to obtain the existence of $\mathcal{P}_m \in \mathcal{PT}(s' \twoheadrightarrow s'[\pi]_{p'})$ such that $|\mathcal{P}|_{\mathbf{OR}} = |\mathcal{P}_m|_{\mathbf{OR}}$. Let $\mathcal{P}_j = \lambda\overline{x_i}.s_j \twoheadrightarrow \lambda\overline{x_i}.s_j$ (**RF**) for all $j \neq m$. Then Lemma 3.3.4 holds for $\mathcal{P}' \equiv \frac{\mathcal{P}_1 \ \ldots \ \mathcal{P}_n}{s \twoheadrightarrow s[\pi]_p}$(**MN**).

□

Chapter 4

Higher-Order Semantics with λ-Abstractions

In this chapter we present the declarative foundations of our semantic framework with λ-abstractions by means of a model-theoretic and a fixed-point semantics, proving basic results for each of them. Model-theoretic semantics denote models of *CPRS* programs as algebras that satisfy the pattern rewrite rules of the program. Fixed-point semantics characterizes the pattern model as the least fixed-point of an operator on algebras, that coincides with the idea of successive refinements. Finally we define *CPRS* program modules and prove that the modular semantics defined for the framework is compositional and fully abstract with respect to the defined operations on modules, which is essential to support program transformations, modular verification, and advanced constraint-solving extensions based on this framework.

4.1 Model-Theoretic Semantics

In this section, we define *models* for the *GHRC* logic presented in Chapter 3 and we establish soundness and completeness results of *GHRC*-provability with respect to semantic validity in models. Moreover, we prove that every *CPRS* $\mathcal{R}$ has a *pattern model* $\mathcal{M}_{\mathcal{R}}$, which can be seen as a generalization of the canonic $\mathcal{C}$*-semantics* for logic programming as defined in [30, 31].

To define a model-theoretic semantics we give some technical notions about *domain theory* [90] which are at the basis of the denotational semantics to make this work more self-contained. The primary motivation for the study of domains, which was initiated by *Dana Scott* in the late 1960s, was the search for a denotational semantics of the lambda calculus [89].

First we give the definition of a *partially ordered set with bottom* and some

auxiliary notions about them, which is a key concept in our framework:

Definition 4.1.1 (Partially ordered sets with bottom)

- *A **partially ordered set** (also called **poset** for short) **with bottom** (generically denoted as $\bot$) is a set S with a partial order $\sqsubseteq$ defined between elements of S such that $\bot \sqsubseteq e$ for all $e \in S$.*
- *Let S be a poset with bottom and $D \subseteq S$. Then D is a **directed set** if and only if for all $x, y \in D$ there exists $z \in D$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$.*
- *Let S be a poset with bottom and $x \in S$. Element x is **totally defined** in S if x is maximal (i.e., there not exists $y \neq x \in S$ such that $y \sqsubseteq x$).*

An example of poset with bottom is the set of partial values $Val(\Sigma_\bot, \mathcal{V})$ with the approximation ordering $\sqsubseteq$, defined in Section 2.3.

The adequate structure to denote defined function and constructor symbols in the signature of a program are *cones*; cones are partially ordered sets with bottom that contain all the approximations of any element in the cone. Ideals are cones that also have the property that for each couple of elements there exists a least upper bound of them.

Definition 4.1.2 (Cones and Ideals) *Let S a partially ordered set with bottom:*

- *The set $A \subseteq S$ is a **cone** if and only if $\bot \in A$ and $\forall x \in A, y \in S. y \sqsubseteq x \Rightarrow y \in A$ (the latter condition it is referred as 'A **is downclosed**').*
- *The **set of cones** of S is denoted as $\mathcal{C}(S)$ and is defined as $\mathcal{C}(S) = \{A \subseteq S \mid A \text{ is a cone}\}$.*
- *A set $I \subseteq S$ is an **ideal** if I is a cone and I is a directed set.*
- *The **set of ideals** of S is denoted as $\mathcal{I}(S)$ and is defined as $\mathcal{C}(S) = \{A \subseteq S \mid A \text{ is an ideal}\}$.*
- *The **ideal completion of S** is denoted as $\bar{S}$ and is a poset with bottom consisting on the set $\mathcal{I}(S)$ with the 'set inclusion partial order' $\subseteq$ and bottom element $\emptyset$.*
- *Let S a poset with partial order $\sqsubseteq$ and $x \in S$. The **ideal generated by** x is denoted as $\langle\, x \,\rangle$ and is defined as $\langle\, x \,\rangle = \{y \in S \mid y \sqsubseteq x\}$.*

We interpret a *CPRS*-program over structures consisting of posets with $\perp$ as carriers, whose elements are thought of finite approximations of possibly infinite values in the poset's ideal completion (see [72]). Moreover, monotonic mappings from elements to cones represent defined function symbol denotations reflecting possible non-determinism, and directed in the case of constructor symbols, that are always trivially deterministic.

Definition 4.1.3 (*GHRC*-algebras) *Given a signature $\Sigma = \Sigma_d \cup \Sigma_c$, a **GHRC-algebra** $\mathcal{A}$ is a structure of the form*

$$\mathcal{A} = \langle\, D_{\mathcal{A}}, \{f^{\mathcal{A}}\}_{f \in \Sigma_d}, \{c^{\mathcal{A}}\}_{c \in \Sigma_c}, \circ^{\mathcal{A}} \,\rangle$$

where:

- *The carrier set $D_{\mathcal{A}}$ is a poset with partial order $\sqsubseteq_{D_{\mathcal{A}}}$ and bottom element $\perp_{\mathcal{A}}$.*
- *For each $f \in \Sigma_d$ with arity$(f) = n$, $f^{\mathcal{A}}$ is a monotonic mapping $D^n_{\mathcal{A}} \to \mathcal{C}(D_{\mathcal{A}})$ that verifies for all total $\gamma \in Subst(\Sigma, \mathcal{V})$:*

$$(f^{\mathcal{A}_{\mathcal{D}}}(\overline{t_n}))\gamma = \{d\gamma \,|\, d \in f^{\mathcal{A}_{\mathcal{D}}}(\overline{t_n})\} \subseteq f^{\mathcal{A}_{\mathcal{D}}}(\overline{t_n}\gamma)$$

- *For each $c \in \Sigma_c$ with arity$(c) = n$, $c^{\mathcal{A}}$ is a monotonic mapping $D^n_{\mathcal{A}} \to \mathcal{I}(D_{\mathcal{A}})$.*
- *The apply operation $\circ^{\mathcal{A}}$ is a non-deterministic binary mapping $D_{\mathcal{A}} \times D_{\mathcal{A}} \to \mathcal{C}(D_{\mathcal{A}})$, written in infix notation, such that $\perp_{\mathcal{A}} \circ^{\mathcal{A}} d = \langle \perp_{\mathcal{A}} \rangle$ for all $d \in D_{\mathcal{A}}$.*

The technical condition in the definition of the denotation of defined function symbols is the so-called *consistency condition* close to the notion of *c-interpretation* considered in [31].

The following definitions taken from [24] show how to *evaluate* λ-terms in *GHRC*-algebras; evaluation of terms is dependant on the actual instantiation of variables, that we define as usual by mean of *valuations*, i.e., mappings from variables to elements in the carrier set of the algebra.

Definition 4.1.4 (Valuations) *Let $\mathcal{A}$ a GHRC-algebra and $\mathcal{V}$ a set of variables:*

- *A **valuation** over $\mathcal{A}$ is any mapping $\eta : \mathcal{V} \to D_{\mathcal{A}}$.*
- *A valuation η is **totally defined** if and only if for all $v \in \mathcal{V}$, $\eta(v)$ is a totally defined element of $D_{\mathcal{A}}$.*

Given a valuation η we can evaluate each λ-term in $\mathcal{A}$ as follows:

Definition 4.1.5 (Evaluation of terms) *Let $\mathcal{A}$ a GHRC-algebra over a signature $\Sigma = \Sigma_d \cup \Sigma_c$, $t \in T(\Sigma_\perp, \mathcal{V})$, and η a valuation of $\mathcal{V}$. The* ***evaluation*** *of t in $\mathcal{A}$ under η is denoted as $[\![t]\!]^{\mathcal{A}}_\eta$ and is defined inductively as follows:*

- $[\![\perp]\!]^{\mathcal{A}}_\eta = \langle \perp_{\mathcal{A}} \rangle$.
- $[\![X]\!]^{\mathcal{A}}_\eta = \langle \eta(X) \rangle$, *if* $X \in \mathcal{V}$.
- $[\![c]\!]^{\mathcal{A}}_\eta = \langle c^{\mathcal{A}} \rangle$, *if* $c \in \Sigma_c$.
- $[\![f]\!]^{\mathcal{A}}_\eta = f^{\mathcal{A}}$, *if* $f \in \Sigma_d$ *and* $arity(f) = 0$.
- $[\![f]\!]^{\mathcal{A}}_\eta = \langle f^{\mathcal{A}} \rangle$, *if* $f \in \Sigma_d$ *and* $arity(f) > 0$.
- $[\![(t\ t')]\!]^{\mathcal{A}}_\eta = [\![t]\!]^{\mathcal{A}}_\eta \circ^{\mathcal{A}} [\![t']\!]^{\mathcal{A}}_\eta$.
- $[\![\lambda \overline{x_k}.\, a(\overline{t_n})]\!]^{\mathcal{A}}_\eta = [\![\, (\cdots (a\ t_1) \cdots t_n)\,]\!]^{\mathcal{A}}_\eta$, *if* $\{\overline{x_k \mapsto d_k}\} \in \eta$ *with* $d_i \in D_{\mathcal{A}}$ *arbitrarily fixed.*

Using *GHRC*-algebras, we are able to extend the model-theoretic results from [38] to our higher-order setting with λ-abstractions. As usual, we interpret reduction statements as inclusions between cones and equality statements asserting the existence of at least one common value as a totally defined approximation. In particular, models in *GHRC* are introduced using the following notions of satisfiability:

Definition 4.1.6 (Models) *Let $\mathcal{R}$ be a CPRS and $\mathcal{A}$ be a GHRC-algebra:*

- *Algebra $\mathcal{A}$* ***satisfies a reduction statement*** *$s \twoheadrightarrow t$ under a valuation η (denoted as $\mathcal{A} \vDash_\eta (s \twoheadrightarrow t)$), if and only if $[\![s]\!]^{\mathcal{A}}_\eta \supseteq [\![t]\!]^{\mathcal{A}}_\eta$.*
- *Algebra $\mathcal{A}$* ***satisfies an equality statement*** *$s == t$ under a valuation η (denoted as $\mathcal{A} \vDash_\eta (s == t)$), if and only if $[\![s]\!]^{\mathcal{A}}_\eta \cap [\![t]\!]^{\mathcal{A}}_\eta$ contains a maximal element in $D_{\mathcal{A}}$.*
- *Algebra $\mathcal{A}$* ***satisfies a pattern rewrite rule*** *$(\pi \to r \Leftarrow C) \in \mathcal{R}$ (denoted as $\mathcal{A} \vDash (\pi \to r \Leftarrow C)$), if and only if $\mathcal{A} \vDash_\eta C$ implies $\mathcal{A} \vDash_\eta (\pi \twoheadrightarrow r)$, for every valuation η.*
- *Algebra $\mathcal{A}$ is a* ***model*** *of $\mathcal{R}$ (denoted as $\mathcal{A} \vDash \mathcal{R}$) if and only if $\mathcal{A}$ satisfies all the pattern rewrite rules in $\mathcal{R}$.*

GHRC-provability is sound and complete with respect to this model-theoretic semantics when we consider totally defined valuations. For each *CPRS* $\mathcal{R}$, we can build a so-called *pattern model* $\mathcal{M}_{\mathcal{R}}$ as a *GHRC*-algebra with carrier set $D_{\mathcal{A}}$ the set of values.

Definition 4.1.7 (Pattern Models) *Let $\mathcal{R}$ be a CPRS with signature Σ; the **pattern model** of $\mathcal{R}$ is denoted as $\mathcal{M}_\mathcal{R}$ and is defined as follows:*

- *The carrier set $D_{\mathcal{M}_\mathcal{R}}$ is the poset $Val(\Sigma_\perp, \mathcal{V})$ of partial terms which are values over Σ, with approximation ordering $\sqsubseteq$ (as defined in Section 2.3) and bottom element $\perp$.*
- *For each $c \in \Sigma_c$ with $arity(c) = n$ and for all $t_i \in Val(\Sigma_\perp, \mathcal{V})$, $c^{\mathcal{M}_\mathcal{R}}(\overline{t_n}) = \langle\, c(\overline{t_n}) \,\rangle$.*
- *For each $f \in \Sigma_d$ with $arity(f) = n$ and for all $t_i \in Val(\Sigma_\perp, \mathcal{V})$, $f^{\mathcal{M}_\mathcal{R}}(\overline{t_n}) = \{\, t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t \,\}$.*
- *The apply operation $t_1 \circ^{\mathcal{M}_\mathcal{R}} t_2 = \langle\, (t_1\ t_2) \,\rangle$, whenever $(t_1\ t_2) \in Val(\Sigma_\perp, \mathcal{V})$.*

To finish this section, we prove that the defined pattern model for any *CPRS* $\mathcal{R}$ is a model of the program, and a result that relates deducibility by means of the *GHRC* proof calculus and satisfiability from the pattern model of $\mathcal{R}$. This result is an extension of an equivalent result (see Theorem 5.2 from [38]) that was originally presented in [24].

Theorem 4.1.8 (Adequateness of $\mathcal{M}_\mathcal{R}$) *Let $\mathcal{R}$ be a CPRS. The pattern model $\mathcal{M}_\mathcal{R}$ is a model of $\mathcal{R}$. Moreover, for any reduction or equality statement φ, the following conditions are equivalent:*

1) $\mathcal{R} \vdash \varphi$.

2) $\mathcal{A} \vDash_\eta \varphi$, for every $\mathcal{A} \vDash \mathcal{R}$ and every totally defined valuation η.

3) $\mathcal{M}_\mathcal{R} \vDash_\varepsilon \varphi$, where ε is the identity valuation.

Proof To prove $\mathcal{M}_\mathcal{R} \vDash \mathcal{R}$, we consider a pattern rewrite rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$ and a value substitution $\theta \in VSubst(\Sigma_\perp, \mathcal{V})$ over $\mathcal{M}_\mathcal{R}$. Assume that $\mathcal{M}_\mathcal{R} \vDash_\theta C$. Then, $\mathcal{R} \vdash C\theta$ by applying Definition 4.1.7 (see the proof of the implication 3) $\Rightarrow$ 1) below for more details). It follows that $\mathcal{R} \vdash (f(\overline{l_n}) \twoheadrightarrow r)\theta$, or equivalently, $\mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow r\theta$, by applying the rule **OR** of the *GHRC* calculus (i.e., $\dfrac{\lambda\overline{x_k}.\, s_1 \twoheadrightarrow l_1^{\updownarrow\overline{x_k}}\theta \cdots \lambda\overline{x_k}.\, s_n \twoheadrightarrow l_n^{\updownarrow\overline{x_k}}\theta \quad C^{\updownarrow\overline{x_k}}\theta \quad r^{\updownarrow\overline{x_k}}\theta \twoheadrightarrow u}{\lambda\overline{x_k}.\, f(\overline{s_n}) \twoheadrightarrow u}$ (**OR**) because $\mathcal{R} \vdash r\theta \twoheadrightarrow r\theta$ trivially holds by **RF**). We conclude that $\{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash r\theta \twoheadrightarrow t\} \subseteq \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow t\}$. This means $[\![\, r \,]\!]_\theta^{\mathcal{M}_\mathcal{R}} = [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} = \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid t \in [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}\} = \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \langle\, t \,\rangle = [\![\, t \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}} \subseteq [\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}_\mathcal{R}}\} = \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{M}_\mathcal{R} \vDash_\varepsilon (r\theta \twoheadrightarrow t)\} \subseteq \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash r\theta \twoheadrightarrow t\} \subseteq \{t \in Val(\Sigma_\perp, \mathcal{V}) \mid \mathcal{R} \vdash f(\overline{l_n})\theta \twoheadrightarrow t\} = [\![\, f(\overline{l_n}) \,]\!]_\theta^{\mathcal{M}_\mathcal{R}}$, by applying again Definition 4.1.7 (and again the proof of the implication 3) $\Rightarrow$ 1)). Therefore, $\mathcal{M}_\mathcal{R} \vDash_\theta (f(\overline{l_n}) \twoheadrightarrow r)$,

and we can conclude that $\mathcal{M}_{\mathcal{R}} \models (f(\overline{l_n}) \to r \Leftarrow C)$. By Definition 4.1.6, $\mathcal{M}_{\mathcal{R}}$ is a *GHRC*-model of $\mathcal{R}$ (i.e., $\mathcal{M}_{\mathcal{R}} \models \mathcal{R}$).

Now we prove the equivalence of 1), 2) and 3) by proving that 1) ⇒ 2), 2) ⇒ 3) and 3) ⇒ 1):

1) ⇒ 2) Assume that $\mathcal{R} \vdash \varphi$ for any reduction or equality statement φ. Let $\mathcal{A} \models \mathcal{R}$ be an arbitrarily fixed totally defined valuation η. Now, we prove that $\mathcal{A} \models_\eta \varphi$ by induction on the length of the *GHRC* proof:

B If $\mathcal{R} \vdash \lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\ \bot$ then $[\![\, \lambda\overline{x_k}.\ \bot \,]\!]^{\mathcal{A}}_\eta = \langle\, \bot_{\mathcal{A}} \,\rangle = \{d \in D_{\mathcal{A}} \mid d \sqsubseteq_{D_{\mathcal{A}}} \bot_{\mathcal{A}}\} = \{\bot_{\mathcal{A}}\}$. Since $[\![\, \lambda\overline{x_k}.\pi \,]\!]^{\mathcal{A}}_\eta$ is a downward closed subset of $D_{\mathcal{A}}$, $\bot_{\mathcal{A}} \in [\![\, \lambda\overline{x_k}.\pi \,]\!]^{\mathcal{A}}_\eta$ and $[\![\, \lambda\overline{x_k}.\pi \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, \lambda\overline{x_k}.\ \bot \,]\!]^{\mathcal{A}}_\eta$. By Definition 4.1.6, we conclude that $\mathcal{A} \models_\eta (\lambda\overline{x_k}.\pi \twoheadrightarrow \lambda\overline{x_k}.\bot)$.

MN If $\mathcal{R} \vdash \lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})$, we can assume that $\mathcal{A} \models_\eta (\lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i)$, for $i = 1, \ldots, n$, by induction hypothesis, where $\{\overline{x_k \mapsto d_k}\} \in \eta$ with $d_i \in D_{\mathcal{A}}$ arbitrarily fixed. By Definition 4.1.6, $[\![\, s_i^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, t_i^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta$. We prove that $[\![\, \lambda\overline{x_k}.a(\overline{s_n}) \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, \lambda\overline{x_k}.a(\overline{t_n}) \,]\!]^{\mathcal{A}}_\eta$. Since $[\![\, \lambda\overline{x_k}.a(\overline{s_n}) \,]\!]^{\mathcal{A}}_\eta = (\cdots([\![\, a \,]\!]^{\mathcal{A}}_\eta \circ^{\mathcal{A}} [\![\, s_1^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) \circ^{\mathcal{A}} \ldots \circ^{\mathcal{A}} [\![\, s_n^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) \supseteq (\cdots([\![\, a \,]\!]^{\mathcal{A}}_\eta \circ^{\mathcal{A}} [\![\, t_1^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) \circ^{\mathcal{A}} \cdots \circ^{\mathcal{A}} [\![\, t_n^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) = [\![\, \lambda\overline{x_k}.a(\overline{t_n}) \,]\!]^{\mathcal{A}}_\eta$, we conclude that $\mathcal{A} \models_\eta (\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n}))$.

RF If $\mathcal{R} \vdash s \twoheadrightarrow s$, trivially $[\![\, s \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, s \,]\!]^{\mathcal{A}}_\eta$, and $\mathcal{A} \models_\eta (s \twoheadrightarrow s)$ by Definition 4.1.6.

OR If $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u$, where $f \in \Sigma_d$ and $u \neq \lambda\overline{x_k}.\bot$, we can consider $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$, $\theta \in \mathit{VSubst}(\Sigma_\bot, \mathcal{V})$, and $\{\overline{x_k \mapsto d_k}\} \in \eta$ with $d_i \in D_{\mathcal{A}}$ arbitrarily fixed. Assume that $\mathcal{A} \models_\eta C^{\downarrow\overline{x_k}}\theta$. Then, with $\rho = \eta\theta$, $\mathcal{A} \models_\rho C^{\downarrow\overline{x_k}}$. Since $\mathcal{A} \models \mathcal{R}$ by hypothesis, we conclude that $\mathcal{A} \models_\rho (f(\overline{l_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow r^{\downarrow\overline{x_k}})$ by Definition 4.1.6, we come to $\mathcal{A} \models_\eta (f(\overline{l_n^{\downarrow\overline{x_k}}}) \twoheadrightarrow r)\theta$, or equivalently, $[\![\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, r^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta$. By induction hypothesis, $[\![\, s_i^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, l_i^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta$, for $i = 1, \ldots, n$, and $[\![\, r^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, u \,]\!]^{\mathcal{A}}_\eta$. We prove that $[\![\, f(\overline{s_n})^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, u \,]\!]^{\mathcal{A}}_\eta$. Since $[\![\, f(\overline{s_n})^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta = (\cdots([\![\, f \,]\!]^{\mathcal{A}}_\eta \circ^{\mathcal{A}} [\![\, s_1^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) \circ^{\mathcal{A}} \ldots \circ^{\mathcal{A}} [\![\, s_n^{\downarrow\overline{x_k}} \,]\!]^{\mathcal{A}}_\eta) \supseteq (\cdots([\![\, f \,]\!]^{\mathcal{A}}_\eta \circ^{\mathcal{A}} [\![\, l_1^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta) \circ^{\mathcal{A}} \cdots \circ^{\mathcal{A}} [\![\, l_n^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta) = [\![\, f(\overline{l_n^{\downarrow\overline{x_k}}\theta}) \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, r^{\downarrow\overline{x_k}}\theta \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, u \,]\!]^{\mathcal{A}}_\eta$, we conclude $\mathcal{A} \models_\eta (\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow u)$.

J If $\mathcal{R} \vdash s == t$, we can assume that $\mathcal{A} \models_\eta (s \twoheadrightarrow u)$ and $\mathcal{A} \models_\eta (t \twoheadrightarrow u)$ for some $u \in \mathit{Val}(\Sigma, \mathcal{V})$, by induction hypothesis. By Definition 4.1.6, this means $[\![\, s \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, u \,]\!]^{\mathcal{A}}_\eta$ and $[\![\, t \,]\!]^{\mathcal{A}}_\eta \supseteq [\![\, u \,]\!]^{\mathcal{A}}_\eta$. We know that $[\![\, u \,]\!]^{\mathcal{A}}_\eta = \langle\, d \,\rangle$ for some maximal element $d \in D_{\mathcal{A}}$. Again, by Definition 4.1.6, we can conclude that $[\![\, s \,]\!]^{\mathcal{A}}_\eta \cap [\![\, s \,]\!]^{\mathcal{A}}_\eta$ contains a maximal element $d \in D_{\mathcal{A}}$, and $\mathcal{A} \models_\eta (s == t)$.

2) $\Rightarrow$ 3) Trivially, because $\mathcal{M}_{\mathcal{R}}$ is a *GHRC*-model of $\mathcal{R}$ and ε is a totally defined valuation.

3) $\Rightarrow$ 1) Let ε be the identity valuation over $\mathcal{M}_{\mathcal{R}}$. For any approximation or equality statement φ, we prove that if $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} \varphi$ then $\mathcal{R} \vdash \varphi$. We reason by induction on the size of φ, defined as the number of symbols occurring in φ. Since $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} \varphi$, there are only five cases to consider:

1. If $\varphi \equiv \lambda\overline{x_k}.\,\pi \twoheadrightarrow \lambda\overline{x_k}.\,\bot$ then $\mathcal{R} \vdash \lambda\overline{x_k}.\,\pi \twoheadrightarrow \lambda\overline{x_k}.\,\bot$ holds because of rule **B**.

2. If $\varphi \equiv s \twoheadrightarrow s$ then $\mathcal{R} \vdash s \twoheadrightarrow s$ holds because of rule **RF**.

3. If $\varphi \equiv \lambda\overline{x_k}.\,a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.\,a(\overline{t_n})$, by construction of $\mathcal{M}_{\mathcal{R}}$ we have that $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} \varphi$ entails $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} (\lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i)$, for $i = 1, \ldots, n$. Then, by induction hypothesis, $\mathcal{R} \vdash \lambda\overline{x_k}.s_i \twoheadrightarrow \lambda\overline{x_k}.t_i$ and $\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow \lambda\overline{x_k}.t_1 \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow \lambda\overline{x_k}.t_n}{\lambda\overline{x_k}.a(\overline{s_n}) \twoheadrightarrow \lambda\overline{x_k}.a(\overline{t_n})}$(**MN**).

4. If $\varphi \equiv \lambda\overline{x_k}.\,f(\overline{s_n}) \twoheadrightarrow t$ with $t \neq \lambda\overline{x_k}.\,\bot$, then $[\![\, t \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}} = \langle\, t\varepsilon \,\rangle = \langle\, t \,\rangle$ and $t \in (\cdots (f^{\mathcal{M}_{\mathcal{R}}} \circ^{\mathcal{M}_{\mathcal{R}}} [\![\, s_1^{\updownarrow \overline{x_k}} \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}}) \circ^{\mathcal{M}_{\mathcal{R}}} \cdots \circ^{\mathcal{M}_{\mathcal{R}}} [\![\, s_n^{\updownarrow \overline{x_k}} \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}})$. Hence, there are some $t_i \in [\![\, s_i^{\updownarrow \overline{x_k}} \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}}$, for $i = 1, \ldots, n$, such that $t \in f^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n})$. Therefore, we have $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} (\lambda\overline{x_k}.s_i \twoheadrightarrow t_i)$ and $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$ by construction of $\mathcal{M}_{\mathcal{R}}$ (see Definition 4.1.7). By induction hypothesis, we can assume $\mathcal{R} \vdash \lambda\overline{x_k}.s_i \twoheadrightarrow t_i$, for $i = 1, \ldots, n$. Then, $\frac{\lambda\overline{x_k}.s_1 \twoheadrightarrow t_1 \cdots \lambda\overline{x_k}.s_n \twoheadrightarrow t_n}{\lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow f(\overline{t_n})}$(**MN**) and $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow f(\overline{t_n})$. Since $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$, we conclude that $\mathcal{R} \vdash \lambda\overline{x_k}.f(\overline{s_n}) \twoheadrightarrow t$.

5. If $\varphi \equiv s == t$, we get that $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} (s == t)$ entails the existence of a maximal element $u \in \mathit{Val}(\Sigma_{\bot}, \mathcal{V})$ such that $u \in [\![\, s \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}} \cap [\![\, t \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}}$. Since maximal elements in $\mathcal{M}_{\mathcal{R}}$ are totally defined elements in $\mathit{Val}(\Sigma_{\bot}, \mathcal{V})$, we have that u is a total value (i.e., $u \in \mathit{Val}(\Sigma_{\bot}, \mathcal{V})$). Then, $u \in [\![\, s \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}}$ and $u \in [\![\, t \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}}$. Due to the fact that $[\![\, u \,]\!]_{\varepsilon}^{\mathcal{M}_{\mathcal{R}}} = \langle\, u\varepsilon \,\rangle = \langle\, u \,\rangle$, we can deduce that $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} (s \twoheadrightarrow u)$ and $\mathcal{M}_{\mathcal{R}} \models_{\varepsilon} (t \twoheadrightarrow u)$. By induction hypothesis, $\mathcal{R} \vdash s \twoheadrightarrow u$ and $\mathcal{R} \vdash t \twoheadrightarrow u$ for $u \in \mathit{Val}(\Sigma_{\bot}, \mathcal{V})$, and then $\frac{s \twoheadrightarrow u \quad t \twoheadrightarrow u}{s == t}$(**J**). We can conclude that $\mathcal{R} \vdash \varphi$.

□

The equivalence between items 1) and 2) shows that the *GHRC* proof calculus is sound and complete for deriving those statements which hold in all models of a given *CPRS* under all possible totally defined valuations:

$$\mathcal{R} \vdash \varphi \Leftrightarrow \mathcal{A} \models_{\eta} \varphi, \text{ for every } \mathcal{A} \models \mathcal{R} \text{ and every totally defined valuation } \eta.$$

4.2 Fixed-Point Semantics

The denotational characterization of a program is usually given in terms of the least fixed-point of a continuous transformation associated to it. In this section we prove, for every *CPRS* $\mathcal{R}$, that the pattern model $\mathcal{M}_\mathcal{R}$ is the least fixed-point of a continuous operator defined over *pattern algebras*, which are *GHRC*-algebras with carrier set the set of partial values with respect to $\mathcal{R}$, ordered by the approximation ordering $\sqsubseteq$ defined in Section 2.3, and a fixed interpretation for data constructors. The pattern model $\mathcal{M}_\mathcal{R}$ of a *CPRS* is a particular case of pattern algebra.

Definition 4.2.1 (Pattern Algebras) *Let $\mathcal{A}$ be a GHRC-algebra and $\mathcal{R}$ a CPRS with signature Σ. The algebra $\mathcal{A}$ is a* ***pattern algebra*** *if and only if:*

- *The carrier set $D_\mathcal{A}$ is the set of partial values $Val(\Sigma_\bot, \mathcal{V}) \subseteq T(\Sigma_\bot, \mathcal{V})$ with partial order $\sqsubseteq_{D_\mathcal{A}}$ the approximation ordering $\sqsubseteq$ defined in Section 2.3.*
- *For each $c \in \Sigma_c$ with $arity(c) = n$ and for all $t_i \in Val(\Sigma_\bot, \mathcal{V})$, $c^{\mathcal{M}_\mathcal{R}}(\overline{t_n}) = \langle\, c(\overline{t_n}) \,\rangle$.*
- *The apply operation $t_1 \circ^{\mathcal{M}_\mathcal{R}} t_2 = \langle\, (t_1\ t_2) \,\rangle$, whenever $(t_1\ t_2) \in Val(\Sigma_\bot, \mathcal{V})$.*

The set of pattern algebras of Σ associated to $\mathcal{R}$ is denoted as Alg_Σ.

The approach we use here is similar to that applied in the field of logic programming [4]. However, the notion of interpretation, and the corresponding mathematical aspects, have to be reformulated in the context of pattern algebras. This approach has been also used in the context of previous formalisms to model higher-order functional logic programming (see e.g., [47]). However, [47] does not deal with some relevant aspects (e.g., non-determinism) of the *GHRC*-programming approach we are considering here.

It is interesting to notice that the set of pattern algebras Alg_Σ of signature Σ associated to a *CPRS* $\mathcal{R}$ is a poset with bottom when defining an ordering based on interpretation of defined function symbols as a set inclusion of cones. Let $\mathcal{A}$ and $\mathcal{B}$ be two pattern algebras of Alg_Σ; we can define the relationship $\mathcal{A} \sqsubseteq \mathcal{B}$ as $f^\mathcal{A}(\overline{t_n}) \subseteq f^\mathcal{B}(\overline{t_n})$ for all $f \in \Sigma_d$, when $arity(f) > 0$, and $f^\mathcal{A} \subseteq f^\mathcal{B}$, when $arity(f) = 0$; this relationship is obviously a partial ordering and the set Alg_Σ of pattern algebras over Σ with the partial ordering $\sqsubseteq$ is a poset. This poset has a bottom element $\bot_\Sigma$ and a top element $\top_\Sigma$ characterized by $f^{\bot_\Sigma}(\overline{t_n}) = \langle\bot\rangle$ and $f^{\top_\Sigma}(\overline{t_n}) = Val(\Sigma_\bot, \mathcal{V})$, respectively, for each $f \in \Sigma_d$ with $arity(f) \geq 0$.

Given a subset S of pattern algebras, the following definitions for each $f \in \Sigma_d$:

- $f^{\sqcup S}(\overline{t_n}) =_{def} \bigcup_{\mathcal{A} \in S} f^\mathcal{A}(\overline{t_n})$

- $f^{\sqcap S}(\overline{t_n}) =_{def} \bigcap_{\mathcal{A}\in S} f^{\mathcal{A}}(\overline{t_n})$

characterize two pattern algebras, $\sqcup S$ and $\sqcap S$, respectively, because the union and intersection of any number of cones are also cones, and the resulting functions in the above definitions are obviously monotonic if $f^{\mathcal{A}}$ is monotonic for all $\mathcal{A} \in S$. Clearly, $\sqcup S$ and $\sqcap S$ are the *least upper bound* and the *greatest lower bound* of S, respectively. So, the set of *GHRC*-pattern algebras is a *complete lattice*.

The following lemma from [24] establishes the continuity of valuations in Alg_Σ, that is, pattern algebras with a less defined interpretation of defined function symbols will evaluate terms to equally or less defined cones.

Lemma 4.2.2 (Continuity of valuations in Alg_Σ)
For each partial term $t \in T(\Sigma_\perp, \mathcal{V})$ and each value substitution $\theta \in VSubst(\Sigma_\perp, \mathcal{V})$:

1. *If $\mathcal{A} \sqsubseteq \mathcal{B}$ then $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$, for $\mathcal{A}, \mathcal{B} \in Alg_\Sigma$.*
2. *$[\![t]\!]^{\sqcup D}_\theta = \bigcup_{\mathcal{A}\in D} [\![t]\!]^{\mathcal{A}}_\theta$, for all directed subsets $D \subseteq Alg_\Sigma$.*

Proof

1. The first statement is proved by induction on the structure of t:

   - If $t \in \{\perp\} \cup \mathcal{V}$ or $t \in \Sigma_c$ with $arity(t) = 0$ then $[\![t]\!]^{\mathcal{A}}_\theta$ does not depend on the particular pattern algebra $\mathcal{A}$ and $[\![t]\!]^{\mathcal{A}}_\theta = [\![t]\!]^{\mathcal{B}}_\theta$.
   - If $t \in \Sigma_d$ with $arity(t) = 0$, $\mathcal{A} \sqsubseteq \mathcal{B}$ implies $t^{\mathcal{A}} \subseteq t^{\mathcal{B}}$ and then $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$.
   - If $t = f(\overline{t_n})$ with $f \in \Sigma$ and $arity(f) = n > 0$, assuming $[\![t_i]\!]^{\mathcal{A}}_\theta \subseteq [\![t_i]\!]^{\mathcal{B}}_\theta$, for $i = 1, \ldots, n$, as the induction hypothesis, for every $s \in [\![t]\!]^{\mathcal{A}}_\theta$ we have $s \in f^{\mathcal{A}}(\overline{s_n})$ for some $s_i \in [\![t_i]\!]^{\mathcal{A}}_\theta$, which implies $s \in f^{\mathcal{B}}(\overline{s_n})$ with $s_i \in [\![t_i]\!]^{\mathcal{B}}_\theta$ as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![t]\!]^{\mathcal{B}}_\theta$, and consequently $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$.
   - If $t = (t_1\ t_2)$, assuming $[\![t_i]\!]^{\mathcal{A}}_\theta \subseteq [\![t_i]\!]^{\mathcal{B}}_\theta$ for $i = 1, 2$, as the induction hypothesis, for every $s \in [\![t]\!]^{\mathcal{A}}_\theta = [\![(t_1\ t_2)]\!]^{\mathcal{A}}_\theta = [\![t_1]\!]^{\mathcal{A}}_\theta \circ^{\mathcal{A}} [\![t_2]\!]^{\mathcal{A}}_\theta \subseteq [\![t_1]\!]^{\mathcal{B}}_\theta \circ^{\mathcal{B}} [\![t_2]\!]^{\mathcal{B}}_\theta = [\![t]\!]^{\mathcal{B}}_\theta$ as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![t]\!]^{\mathcal{B}}_\theta$, and consequently $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$.
   - If $t = \lambda\overline{x}_k.a(\overline{t_n})$, assuming $[\![a]\!]^{\mathcal{A}}_\theta \subseteq [\![a]\!]^{\mathcal{B}}_\theta$ and $[\![t_i]\!]^{\mathcal{A}}_\theta \subseteq [\![t_i]\!]^{\mathcal{B}}_\theta$ for $i = 1, \ldots, n$, as the induction hypothesis, where $\{\overline{x_k \mapsto s_k}\} \in \theta$ with s_i arbitrarily fixed partial values of $D_{\mathcal{A}}$, for every $s \in [\![t]\!]^{\mathcal{A}}_\theta = [\![\lambda\overline{x}_k.a(\overline{t_n})]\!]^{\mathcal{A}}_\theta = [\![(\cdots (a\ t_1) \cdots t_n)]\!]^{\mathcal{A}}_\theta = (\cdots ([\![a]\!]^{\mathcal{A}}_\theta \circ^{\mathcal{A}} [\![t_1]\!]^{\mathcal{A}}_\theta) \circ^{\mathcal{A}} \ldots \circ^{\mathcal{A}} [\![t_n]\!]^{\mathcal{A}}_\theta) \subseteq (\cdots ([\![a]\!]^{\mathcal{B}}_\theta \circ^{\mathcal{B}} [\![t_1]\!]^{\mathcal{B}}_\theta) \circ^{\mathcal{B}} \cdots \circ^{\mathcal{B}} [\![t_n]\!]^{\mathcal{B}}_\theta) = [\![(\cdots (a\ t_1) \cdots t_n)]\!]^{\mathcal{B}}_\theta = [\![\lambda\overline{x}_k.a(\overline{t_n})]\!]^{\mathcal{A}}_\theta = [\![t]\!]^{\mathcal{A}}_\theta$, as a consequence of $\mathcal{A} \sqsubseteq \mathcal{B}$ and the induction hypothesis. Thus, we get $s \in [\![t]\!]^{\mathcal{B}}_\theta$, and consequently $[\![t]\!]^{\mathcal{A}}_\theta \subseteq [\![t]\!]^{\mathcal{B}}_\theta$.

2. To prove the second statement we only need to prove the following inclusion: $[\![t]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$, because the inclusion in the other way is trivially derived from the first statement. We also proceed by induction on t:

   - If $t \in \{\bot\} \cup \mathcal{V}$ or $t \in \Sigma_c$ with $arity(t) = 0$ then, as $[\![t]\!]_\theta^{\mathcal{A}}$ does not depend on $\mathcal{A}$, $[\![t]\!]_\theta^{\sqcup D} = [\![t]\!]_\theta^{\mathcal{A}}$ for all $\mathcal{A} \in D$.
   - If $t \in \Sigma_d$ with $arity(t) = 0$ then $[\![t]\!]_\theta^{\sqcup D} = t^{\sqcup D}$ and, by definition, $t^{\sqcup D} = \bigcup_{\mathcal{A}\in D} t^{\mathcal{A}}$. So, in all these cases, $[\![t]\!]_\theta^{\sqcup D} = \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$.
   - If $t = f(\overline{t_n})$ with $f \in \Sigma_d$ and $arity(f) = n > 0$, assuming $[\![t_i]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t_i]\!]_\theta^{\mathcal{A}}$, $i = 1, \ldots, n$, as the induction hypothesis, for every $s \in [\![t]\!]_\theta^{\sqcup D}$ we have $s \in f^{\sqcup D}(\overline{s_n})$ for some $s_i \in [\![t_i]\!]_\theta^{\sqcup D}$, $i = 1, \ldots, n$. By definition, $f^{\sqcup D}(\overline{s_n}) = \bigcup_{\mathcal{A}\in D} f^{\mathcal{A}}(\overline{s_n})$, and from this and the induction hypothesis we can deduce $s \in f^{\mathcal{A}_0}(\overline{s_n})$ with $s_i \in [\![t_i]\!]_\theta^{\mathcal{A}_i}$, for some $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n \in D$. Since D is directed, there exists $\mathcal{A} \in D$, such that $\mathcal{A}_i \sqsubseteq \mathcal{A}$, $i = 0, 1, \ldots, n$, and so $s \in f^{\mathcal{A}}(\overline{s_n})$ with $s_i \in [\![t_i]\!]_\theta^{\mathcal{A}}$, which implies $s \in [\![t]\!]_\theta^{\mathcal{A}}$ and $[\![t]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$.
   - If $t = (t_1\ t_2)$, we know by definition that $\circ^{\sqcup D} = \bigcup_{\mathcal{A}\in D} \circ^{\mathcal{A}}$, and D is directed by initial hypothesis. Assuming $[\![t_i]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t_i]\!]_\theta^{\mathcal{A}}$, $i = 1, 2$, as the induction hypothesis, for every $s \in [\![t]\!]_\theta^{\sqcup D} = [\![(t_1\ t_2)]\!]_\theta^{\sqcup D} = [\![t_1]\!]_\theta^{\sqcup D} \circ^{\sqcup D} [\![t_2]\!]_\theta^{\sqcup D} \subseteq (\bigcup_{\mathcal{A}\in D}[\![t_1]\!]_\theta^{\mathcal{A}}) \circ^{\sqcup D} (\bigcup_{\mathcal{A}\in D}[\![t_2]\!]_\theta^{\mathcal{A}}) = \bigcup_{\mathcal{A}\in D}([\![t_1]\!]_\theta^{\mathcal{A}} \circ^{\mathcal{A}} [\![t_2]\!]_\theta^{\mathcal{A}}) = \bigcup_{\mathcal{A}\in D}[\![(t_1\ t_2)]\!]_\theta^{\mathcal{A}} = \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$, which implies $s \in \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$ and $[\![t]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$.
   - If $t = \lambda\overline{x_k}.a(\overline{t_n})$, assuming $[\![a]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![a]\!]_\theta^{\mathcal{A}}$ and $[\![t_i]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D} [\![t_i]\!]_\theta^{\mathcal{A}}$, for $i = 1, \ldots, n$, as the induction hypothesis, where $\{\overline{x_k \mapsto s_k}\} \in \theta$ with s_i arbitrarily fixed partial values of $\mathcal{A} \in D$ and D directed, for every $s \in [\![t]\!]_\theta^{\sqcup D} = [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]_\theta^{\sqcup D} = [\![(\cdots(a\ t_1)\cdots t_n)]\!]_\theta^{\sqcup D} = (\cdots([\![a]\!]_\theta^{\sqcup D} \circ^{\sqcup D} [\![t_1]\!]_\theta^{\sqcup D}) \circ^{\sqcup D} \cdots \circ^{\sqcup D} [\![t_n]\!]_\theta^{\sqcup D}) \subseteq (\cdots((\bigcup_{\mathcal{A}\in D}[\![a]\!]_\theta^{\mathcal{A}}) \circ^{\sqcup D} (\bigcup_{\mathcal{A}\in D}[\![t_1]\!]_\theta^{\mathcal{A}})) \circ^{\sqcup D} \cdots \circ^{\sqcup D} (\bigcup_{\mathcal{A}\in D}[\![t_n]\!]_\theta^{\mathcal{A}})) = (\cdots(\bigcup_{\mathcal{A}\in D}([\![a]\!]_\theta^{\mathcal{A}} \circ^{\mathcal{A}} [\![t_1]\!]_\theta^{\mathcal{A}})) \circ^{\sqcup D} \cdots \circ^{\sqcup D} (\bigcup_{\mathcal{A}\in D}[\![t_n]\!]_\theta^{\mathcal{A}})) = \bigcup_{\mathcal{A}\in D}(\cdots([\![a]\!]_\theta^{\mathcal{A}} \circ^{\mathcal{A}} [\![t_1]\!]_\theta^{\mathcal{A}}) \circ^{\mathcal{A}} \cdots \circ^{\mathcal{A}} [\![t_n]\!]_\theta^{\mathcal{A}}) = \bigcup_{\mathcal{A}\in D} [\![\lambda\overline{x_k}.a(\overline{t_n})]\!]_\theta^{\mathcal{A}} = \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$, which implies $s \in \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$ and $[\![t]\!]_\theta^{\sqcup D} \subseteq \bigcup_{\mathcal{A}\in D}[\![t]\!]_\theta^{\mathcal{A}}$.

□

Given a *CPRS* $\mathcal{R}$ with signature Σ, we can define a *pattern algebra transformer* $\mathbb{T}_\mathcal{R} : Alg_\Sigma \to Alg_\Sigma$, similar to the immediate consequences operator used in logic programming [30, 31], by fixing the interpretation of each function symbol $f \in \Sigma_d$ in the transformed pattern algebra $\mathbb{T}_\mathcal{R}(\mathcal{A})$, as the result of one step application of those pattern rewrite rules of $\mathcal{R}$ defining f, satisfied in $\mathcal{A} \in Alg_\Sigma$. We formalize this idea by defining, for each $f \in \Sigma_d$:

$$f^{\mathbb{T}_{\mathcal{R}}(\mathcal{A})}(\overline{t_n}) =_{def} \{\bot\} \cup \{\, t \mid t \in [\![f(\overline{t_n})]\!]^{\mathcal{A}}_{\varepsilon} \,\} \quad \cup \quad \{\, t \mid \exists (f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_{\bot}, \; l_i \sqsubseteq t_i, \mathcal{A} \models_{\varepsilon} C, t \in [\![r]\!]^{\mathcal{A}}_{\varepsilon} \,\}$$

This is basically a union of cones. This definition corresponds to a monotonic mapping because all rule instances $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_{\bot}$, applicable to arguments $\overline{t'_n}$, are also applicable to arguments $\overline{t_n}$ such that $t'_i \sqsubseteq t_i$, for $i = 1, \ldots, n$, and so the corresponding interpretation characterizes a pattern algebra. From this definition of $\mathbb{T}_{\mathcal{R}}$ we can easily derive the continuity of the operator $\mathbb{T}_{\mathcal{R}}$ in Alg_{Σ} with the help of some auxiliary lemmas as presented in [23, 24].

Lemma 4.2.3 (Upper Bound) *Let C be a finite set of equality statements and D a directed subset of Alg_{Σ}. Then, $\sqcup D \models_{\eta} C$ implies that there exists $\mathcal{A} \in D$ such that $\mathcal{A} \models_{\eta} C$.*

Proof It is sufficient to prove that this lemma holds when C reduces to one equality statement $s == t$, because with more statements we shall obtain pattern algebras $\mathcal{A}_1, \ldots, \mathcal{A}_n$, one for each equality statement, and the upper bound $\sqcup\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ will satisfy all equality statements in C. By definition, $\sqcup D \models_{\eta} (s == t)$ implies that there exists a totally defined $r \in [\![s]\!]^{\sqcup D}_{\eta} \cap [\![t]\!]^{\sqcup D}_{\eta}$, and by the second statement of Lemma 4.2.2, if $r \in [\![s]\!]^{\sqcup D}_{\eta}$ then $r \in [\![s]\!]^{\mathcal{A}_1}_{\eta}$ for some $\mathcal{A}_1 \in D$, and if $r \in [\![t]\!]^{\sqcup D}_{\eta}$ then $r \in [\![t]\!]^{\mathcal{A}_2}_{\eta}$ for some $\mathcal{A}_2 \in D$. By the first statement of Lemma 4.2.2, considering $\mathcal{A} \in D$ such that $\mathcal{A}_i \sqsubseteq \mathcal{A}$, $i = 1, 2$, we have a pattern algebra such that $r \in [\![s]\!]^{\mathcal{A}}_{\eta} \cap [\![t]\!]^{\mathcal{A}}_{\eta}$, and consequently $\mathcal{A} \models_{\eta} (s == t)$.

□

Lemma 4.2.4 (Continuity of $\mathbb{T}_{\mathcal{R}}$) *For each CPRS $\mathcal{R}$ its associated operator $\mathbb{T}_{\mathcal{R}}$ is continuous.*

Proof *$\mathbb{T}_{\mathcal{R}}$ is monotonic.* Given $\mathcal{A}, \mathcal{B} \in Alg_{\Sigma}$ such that $\mathcal{A} \sqsubseteq \mathcal{B}$, if $\mathcal{A} \models_{\varepsilon} C$ then $\mathcal{B} \models_{\varepsilon} C$ for every set C of equality statements, and by the first statement of Lemma 4.2.2, $[\![t]\!]^{\mathcal{A}}_{\varepsilon} \subseteq [\![t]\!]^{\mathcal{B}}_{\varepsilon}$ for every term t; hence, every rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_{\bot}$ applicable to obtain $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{A})}(\overline{t_n})$ also will be applicable to obtain $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{B})}(\overline{t_n})$, and therefore $\mathbb{T}_{\mathcal{R}}(\mathcal{A}) \sqsubseteq \mathbb{T}_{\mathcal{R}}(\mathcal{B})$. *$\mathbb{T}_{\mathcal{R}}$ is continuous.* For every directed set $D \subseteq Alg_{\Sigma}$, $\mathbb{T}_{\mathcal{R}}(\sqcup D) \sqsubseteq \sqcup\{\mathbb{T}_{\mathcal{R}}(\mathcal{A}) \mid \mathcal{A} \in D\}$ because each rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_{\bot}$ that is applicable to obtain $f^{\mathbb{T}_{\mathcal{R}}(\sqcup D)}(\overline{t_n})$, by Lemma 4.2.2 and Lemma 4.2.3, is also applicable to obtain $\bigcup_{\mathcal{A} \in D} f^{\mathbb{T}_{\mathcal{R}}(\mathcal{A})}(\overline{t_n})$, and this expression is $f^{\sqcup\{\mathbb{T}_{\mathcal{R}}(\mathcal{A}) \mid \mathcal{A} \in D\}}(\overline{t_n})$. The inclusion in the other way is trivial.

□

Thus, $\mathbb{T}_{\mathcal{R}}$ has a *least fixpoint* $\mathfrak{F}_{\mathcal{R}}$ given by $\sqcup A_{\mathcal{R}}$ (that is also the *least pre-fixpoint*), where $A_{\mathcal{R}}$ is the chain of pattern algebras $\mathcal{A}_i$, $i \in \mathbb{N}$, such that:

$$\mathcal{A}_0 = \bot_{\Sigma} \sqsubseteq \cdots \sqsubseteq \mathcal{A}_{i+1} = \mathbb{T}_{\mathcal{R}}(\mathcal{A}_i) \sqsubseteq \cdots$$

$\mathfrak{F}_{\mathcal{R}}$ is also denoted as $\mathbb{T}_{\mathcal{R}}^{\omega}(\bot_{\Sigma})$ (see [1]). From these notions and with the help of some auxiliary lemmas, we can finally prove that $\mathfrak{F}_{\mathcal{R}}$ coincides with $\mathcal{M}_{\mathcal{R}}$.

Lemma 4.2.5 (Model Characterization) *Given a CPRS $\mathcal{R}$, a pattern algebra $\mathcal{M}$ is a GHRC-model of $\mathcal{R}$ if and only if $\mathbb{T}_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{M}$.*

Proof First, we will prove that $\mathbb{T}_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for each pattern algebra $\mathcal{M}$ which is a *GHRC*-model of $\mathcal{R}$. Let us consider $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{M})}(\overline{t_n})$ for $f \in \Sigma_d$ with $arity(f) = n > 0$ and $t_i \in Val(\Sigma_\bot, \mathcal{V})$ for $i = 1, \ldots, n$. If there exists a rule instance $(f(\overline{l_n}) \to r \Leftarrow C) \in [\mathcal{R}]_\bot$ with $r \neq \bot$, $l_i \sqsubseteq t_i$, and $\mathcal{M} \vDash_\varepsilon C$, then, as $\mathcal{M}$ is a *GHRC*-model of $\mathcal{R}$, $[\![\, r \,]\!]_\varepsilon^{\mathcal{M}} \subseteq [\![\, f(\overline{l_n}) \,]\!]_\varepsilon^{\mathcal{M}}$. Then, $[\![\, f(\overline{l_n}) \,]\!]_\varepsilon^{\mathcal{M}} = f^{\mathcal{M}}(\overline{l_n})$, and by $f^{\mathcal{M}}$ monotonic, $f^{\mathcal{M}}(\overline{l_n}) \subseteq f^{\mathcal{M}}(\overline{t_n})$, and $[\![\, r \,]\!]_\varepsilon^{\mathcal{M}} \subseteq f^{\mathcal{M}}(\overline{t_n})$. Thus, $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{M})}(\overline{t_n}) \subseteq f^{\mathcal{M}}(\overline{t_n})$, and consequently, $\mathbb{T}_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{M}$. For $f \in \Sigma_d$ with $arity(f) = 0$, the proof is simpler.

Now, we will prove that every pattern algebra $\mathcal{M}$ such that $\mathbb{T}_{\mathcal{R}}(\mathcal{M}) \sqsubseteq \mathcal{M}$ is a *GHRC*-model of $\mathcal{R}$. Given a rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$, for $\theta \in VSubst(\Sigma_\bot, \mathcal{V})$ such that $\mathcal{M} \vDash_\varepsilon C\theta$, or equivalently, $\mathcal{M} \vDash_\theta C$, we can consider $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{M})}(\overline{t_n\theta})$, and because of the instance $(f(\overline{l_n}) \to r \Leftarrow C)\theta \in [\mathcal{R}]_\bot$, we have $[\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}} \subseteq f^{\mathbb{T}_{\mathcal{R}}(\mathcal{M})}(\overline{t_n\theta})$. By initial hypothesis, $f^{\mathbb{T}_{\mathcal{R}}(\mathcal{M})}(\overline{t_n\theta}) \subseteq f^{\mathcal{M}}(\overline{t_n\theta})$, $[\![\, r\theta \,]\!]_\varepsilon^{\mathcal{M}} = [\![\, r \,]\!]_\theta^{\mathcal{M}}$, and $f^{\mathcal{M}}(\overline{t_n\theta}) = [\![\, f(\overline{t_n}) \,]\!]_\theta^{\mathcal{M}}$; thus $[\![\, r \,]\!]_\theta^{\mathcal{M}} \subseteq [\![\, f(\overline{t_n}) \,]\!]_\theta^{\mathcal{M}}$, which is $\mathcal{M} \vDash_\theta (f(\overline{l_n}) \twoheadrightarrow r)$, and then $\mathcal{M}$ satisfies the rule $(f(\overline{l_n}) \to r \Leftarrow C) \in \mathcal{R}$.

□

Lemma 4.2.6 (Semantic Characterization) *Given a CPRS $\mathcal{R}$, $s \in \mathcal{T}(\Sigma_\bot, \mathcal{V})$, and $t \in Val(\Sigma_\bot, \mathcal{V})$, if $\mathcal{R} \vdash s \twoheadrightarrow t$ then $\mathcal{A}_i \vDash_\varepsilon (s \twoheadrightarrow t)$, for some $\mathcal{A}_i \in A_{\mathcal{R}}$.*

Proof As $\mathbb{T}_{\mathcal{R}}(\sqcup A_{\mathcal{R}}) = \sqcup A_{\mathcal{R}}$, by the *Model Characterization Lemma* (Lemma 4.2.5), $\sqcup A_{\mathcal{R}}$ will be a *GHRC*-model of $\mathcal{R}$. Thus, by Theorem 4.1.8, $\mathcal{R} \vdash s \twoheadrightarrow t$ implies $\sqcup A_{\mathcal{R}} \vDash_\varepsilon (s \twoheadrightarrow t)$, or by Definition 4.1.6, $\langle\, t \,\rangle \subseteq [\![\, s \,]\!]_\varepsilon^{\sqcup A_{\mathcal{R}}}$, that is equivalent to $t \in [\![\, s \,]\!]_\varepsilon^{\sqcup A_{\mathcal{R}}}$. By the second statement of Lemma 4.2.2, $[\![\, s \,]\!]_\varepsilon^{\sqcup A_{\mathcal{R}}} = \bigcup_{\mathcal{A}_i \in A_{\mathcal{R}}} [\![\, s \,]\!]_\varepsilon^{\mathcal{A}_i}$, so there will be an $\mathcal{A}_i \in A_{\mathcal{R}}$ such that $t \in [\![s]\!]_\varepsilon^{\mathcal{A}_i}$, that means $\mathcal{A}_i \vDash_\varepsilon (s \twoheadrightarrow t)$.

□

Theorem 4.2.7 (Fixpoint Characterization of the Least Model) *For every given CPRS $\mathcal{R}$, $\mathcal{M}_{\mathcal{R}}$ is the least fixpoint (and the least pre-fixpoint) of $\mathbb{T}_{\mathcal{R}}$.*

Proof First, we can prove $\sqcup A_{\mathcal{R}} \sqsubseteq \mathcal{M}_{\mathcal{R}}$, from $\mathcal{A}_0 \sqsubseteq \mathcal{M}_{\mathcal{R}}$, $\mathbb{T}_{\mathcal{R}}(\mathcal{M}_{\mathcal{R}}) \sqsubseteq \mathcal{M}_{\mathcal{R}}$ (because $\mathcal{M}_{\mathcal{R}}$ is a model of $\mathcal{R}$ and Lemma 4.2.5), and the continuity of $\mathbb{T}_{\mathcal{R}}$ from 4.2.4 that assures $\mathcal{A}_i \sqsubseteq \mathcal{M}_{\mathcal{R}}$ for all i. Now, we can prove that $\mathcal{M}_{\mathcal{R}} \sqsubseteq \sqcup A_{\mathcal{R}}$ by proving, for each $f \in \Sigma_d$, that $f^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n}) \subseteq f^{\sqcup A_{\mathcal{R}}}(\overline{t_n})$, for $t_1, \ldots, t_n \in Val(\Sigma_\bot, \mathcal{V})$. This inclusion is proved as follows: by Definition 4.1.7, $t \in f^{\mathcal{M}_{\mathcal{R}}}(\overline{t_n})$ is equivalent to $\mathcal{R} \vdash f(\overline{t_n}) \twoheadrightarrow t$,

and this implies $\mathcal{A}_i \models_\varepsilon (f(\overline{t_n}) \twoheadrightarrow t)$ for some $\mathcal{A}_i \in A_\mathcal{R}$ by Lemma 4.2.6. Taking into account that, $[\![f(\overline{t_n})]\!]_\varepsilon^{\mathcal{A}_i} = f^{\mathcal{A}_i}(\overline{t_n})$, and we obtain $t \in f^{\mathcal{A}_i}(\overline{t_n})$. Then, $t \in f^{\sqcup A_\mathcal{R}}(\overline{t_n})$. □

Thus, if we consider the meaning of a *CPRS* $\mathcal{R}$ as the least fixpoint of its associated transformer $\mathbb{T}_\mathcal{R}$, then this fixpoint semantics coincides with the model-theoretic semantics as it happens in logic programming [34]. In fact, this semantics would correspond to the *C-semantics* given in [31].

4.3 Modular Semantics

In the last section of this chapter, we propose an extension of our higher-order declarative programming framework with λ-abstractions to support modular programming by means of the definition of program modules and modular operations between them. A program module is a *CPRS* program that defines an exported signature and imports some defined function symbols from other modules. We also prove that the semantics defined in Sections 4.1 and 4.2 present a good behavior with respect to this extension that makes them suitable to be the theoretical basis of our modular programming approach. This modular extension to the generalized framework of *constraint domains* has been accepted for publication in [27].

4.3.1 Program Modules

In this section we define a modular design for *CPRS* programs. Modules consist on *CPRS* programs that can invoke some defined function symbols in Σ_d but have no rules defining those symbols, which are in other modules. For simplicity, we represent modules as tuples containing the program and the explicit imported and exported signatures.

Definition 4.3.1 (Modules) *Let Σ be a signature. A **CPRS-module** over Σ is a tuple:*

$$\mathcal{M} = \langle\, \Sigma^\mathcal{M}, \Sigma_d^\mathcal{M}, \mathcal{P}_\mathcal{M} \,\rangle$$

where:

- *$\mathcal{P}_\mathcal{M}$ is a CPRS-program.*
- *$\Sigma^\mathcal{M} \subseteq \Sigma$ are the function symbols $g \in \Sigma$ with no definition rule in $\mathcal{P}_\mathcal{M}$ that are invoked in $\mathcal{P}_\mathcal{M}$.*
- *$\Sigma_d^\mathcal{M} \subseteq \Sigma_d$ are the defined function symbols $f \in \Sigma_d$ in $\mathcal{P}_\mathcal{M}$.*

With this definition, $\mathcal{P}_\mathcal{M}$ is the *body* of the module and $\langle\, \Sigma^\mathcal{M}, \Sigma_d^\mathcal{M} \,\rangle$ its *interface*. More precisely, $\Sigma_d^\mathcal{M}$ is the exported signature of defined function symbols and $\Sigma^\mathcal{M}$ is the parameter (imported) signature. With this definition, the interface of a module can be inferred from its body since all the symbols that are defined in the program are exported and every symbol that is invoked but not defined is imported.

Example 4.3.2 (*CPRS*-modules) *Let $\mathcal{P}$ the CPRS of Example 2.3.9. We can write a module $\mathcal{M}_1$ with the differentiation function without defining the operations on natural numbers, that are relegated to another module $\mathcal{M}_2$. Doing this, it is possible to develop more efficient arithmetic operations and use it in the differentiation module just by changing the module that is imported:*

1. $\mathcal{M}_1 = \langle\, \Sigma^{\mathcal{M}_1}, \Sigma_d^{\mathcal{M}_1}, \mathcal{P}_{\mathcal{M}_1} \,\rangle$

   - $\Sigma^{\mathcal{M}_1} = \{add, mult\}$
   - $\Sigma_d^{\mathcal{M}_1} = \{\mathit{diff}\}$
   - $\mathcal{P}_{\mathcal{M}_1} = \{$

   $$\begin{array}{lcl} \mathit{diff}\,(\lambda u.F, x) & \to & zero \\ \mathit{diff}\,(\lambda u.u, x) & \to & s(zero) \\ \mathit{diff}\,(\lambda u.add(F(u), G(u)), x) & \to & add(\mathit{diff}\,(\lambda u.F(u), x), \mathit{diff}\,(\lambda u.G(u), x)) \\ \mathit{diff}\,(\lambda u.mult(F(u), G(u)), x) & \to & add(mult(\mathit{diff}\,(\lambda u.F(u), x), G(x)), \\ & & \quad mult(\mathit{diff}\,(\lambda u.G(u), x), F(x))) \end{array}$$

   $\}$

2. $\mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_2}, \Sigma_d^{\mathcal{M}_2}, \mathcal{P}_{\mathcal{M}_2} \,\rangle$

   - $\Sigma^{\mathcal{M}_2} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_2} = \{add, mult\}$
   - $\mathcal{P}_{\mathcal{M}_2} = \{$

   $$\begin{array}{lcl} add(x, zero) & \to & x \\ add(x, s(y)) & \to & s(add(x, y)) \\ \\ mult(x, zero) & \to & zero \\ mult(x, s(y)) & \to & add(x, mult(x, y)) \end{array}$$

   $\}$

Our modular framework defined for higher-order functional logic programming consists of a small number of operations over *GHRC* modules; we present a set of basic operations that allow us to express typical features of modularization techniques in declarative programming by means of successive applications of these operations. We focus on the most basic composition operations over declarative programs, the *union* of two modules and the *deletion* of a signature in a module.

- **Union of modules.** The union of modules reflects the majority of programming systems that allow adding program definitions stored in several files to the executable code. We define the union of two modules as the module obtained as the simple union of signatures and rules of the two input modules. Given two modules $\mathcal{M}_1 = \langle\, \Sigma^{\mathcal{M}_1}, \Sigma_d^{\mathcal{M}_1}, \mathcal{P}_{\mathcal{M}_1} \,\rangle$ and $\mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_2}, \Sigma_d^{\mathcal{M}_2}, \mathcal{P}_{\mathcal{M}_2} \,\rangle$, their union $\mathcal{M}_1 \cup \mathcal{M}_2$ is defined as the module:

$$\mathcal{M}_1 \cup \mathcal{M}_2 =_{def} \langle (\Sigma^{\mathcal{M}_1} \cup \Sigma^{\mathcal{M}_2}) \setminus (\Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}), \Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}, \mathcal{P}_{\mathcal{M}_1} \cup \mathcal{P}_{\mathcal{M}_2} \,\rangle$$

  Each argument in this operation is considered as an open *CPRS*-program that can be extended or completed with the other argument, possibly with additional rules for its exported signature.

- **Deletion of a signature.** Deletion of a signature Σ in a module $\mathcal{M}$ removes all rules defining function symbols Σ in the signature $\Sigma_d^{\mathcal{M}}$, but maintains the occurrences of these symbols in the right-hand side of rules corresponding to other defined function symbols. This operation can be used to abstract a signature Σ from a module $\mathcal{M}$, and is very useful for making generic modules from concrete ones. Formally, given a module $\mathcal{M} = \langle\, \Sigma^{\mathcal{M}}, \Sigma_d^{\mathcal{M}}, \mathcal{P}_{\mathcal{M}} \,\rangle$, the deletion in $\mathcal{M}$ of a signature Σ of function symbols produces the module:

$$\mathcal{M}\backslash\Sigma =_{def} \langle\, \Sigma'^{\mathcal{M}}, \Sigma_d^{\mathcal{M}} \setminus \Sigma, \mathcal{P}_{\mathcal{M}}\backslash\Sigma \,\rangle$$

  In this definition of $\mathcal{M}\backslash\Sigma$, $\mathcal{P}_{\mathcal{M}}\backslash\Sigma_d$ denotes the set of those rules in $\mathcal{P}_{\mathcal{M}}$ defining function symbols not appearing in Σ, and $\Sigma'^{\mathcal{M}}$ denotes the corresponding parameter signature with all the symbols invoked but not defined in $\mathcal{P}_{\mathcal{M}}\backslash\Sigma$.

The high expressiveness of these operations enable us to model typical constructs for program modularization like *export/import*, *instantiation*, and *inheritance with overriding* in a simple way.

- **Inheritance.** From the union and deletion operations we can model an inheritance relationship between modules. *Inheritance with overriding* may be captured by means of union and deletion as follows:

$$\mathcal{M} \ll \mathcal{N} =_{def} \mathcal{M} \cup (\mathcal{N} \setminus \Sigma_d^{\mathcal{M}})$$

  This new module $\mathcal{M} \ll \mathcal{N}$ inherits all functions in $\mathcal{N}$, with their rules, not defined in $\mathcal{M}$, and uses the rules of $\mathcal{M}$ for all functions defined in $\mathcal{M}$, overriding the definition rules in $\mathcal{N}$. In this case, overriding is carried out by deleting the common signature of the inherited module before adding it to the derived module.

- **Instantiation.** We can instantiate function symbols of a module $\mathcal{M} = \langle\, \Sigma^{\mathcal{M}}, \Sigma_d^{\mathcal{M}}, \mathcal{P}_{\mathcal{M}} \,\rangle$ with function symbols exported by other module, simply by renaming suitably the functions of $\mathcal{M}$ to fit (a part of) the exported signature of $\mathcal{N}$. Thus, we obtain an *instantiation operation* that we denote $\mathcal{M}[\mathcal{N}, \theta]$ and define as:

$$\mathcal{M}[\mathcal{N}, \theta] =_{def} \mathcal{N} \ll \theta(\mathcal{M})$$

  In the definition of $\mathcal{M}[\mathcal{N}, \theta]$, θ is the function symbol renaming substitution that characterizes the instantiation, and $\theta(\mathcal{M}) =_{def} \langle\, \theta(\Sigma^{\mathcal{M}}) \setminus \theta(\Sigma_d^{\mathcal{M}}), \theta(\Sigma_d^{\mathcal{M}}), \mathcal{P}_{\mathcal{M}}\theta \,\rangle$. The renaming operation allows us to change function symbols with other function symbols in a given signature.

Example 4.3.3 (Operations with modules) *In this example we show how each of the four defined operations with modules work considering $\mathcal{M}_1$ and $\mathcal{M}_2$ from Example 4.3.2, $\Sigma_1 = \{add\}$, $\theta = \{add \mapsto addswap\}$ and two modules $\mathcal{M}_3$ and $\mathcal{M}_4$ that define the addition operation by recursion on its first argument that are identical except because they employ different names for the operation:*

1. $\mathcal{M}_3 = \langle\, \Sigma^{\mathcal{M}_3}, \Sigma_d^{\mathcal{M}_3}, \mathcal{P}_{\mathcal{M}_3} \,\rangle$

   - $\Sigma^{\mathcal{M}_3} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_3} = \{add\}$
   - $\mathcal{P}_{\mathcal{M}_3} = \{$

     $$\begin{array}{lcl} add(zero, x) & \rightarrow & x \\ add(s(y), zero) & \rightarrow & s(add(y, x)) \end{array}$$

     $\}$

2. $\mathcal{M}_4 = \langle\, \Sigma^{\mathcal{M}_4}, \Sigma_d^{\mathcal{M}_4}, \mathcal{P}_{\mathcal{M}_4} \,\rangle$

   - $\Sigma^{\mathcal{M}_4} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_4} = \{addswap\}$
   - $\mathcal{P}_{\mathcal{M}_4} = \{$

     $$\begin{array}{lcl} addswap(zero, x) & \rightarrow & x \\ addswap(s(y), zero) & \rightarrow & s(addswap(y, x)) \end{array}$$

     $\}$

Now there is an example of the application of the four operations defined with modules by means of the modules defined before:

1. $\mathcal{M}_1 \cup \mathcal{M}_2 = \langle\, \Sigma^{\mathcal{M}_1 \cup \mathcal{M}_2}, \Sigma_d^{\mathcal{M}_1 \cup \mathcal{M}_2}, \mathcal{P}_{\mathcal{M}_1 \cup \mathcal{M}_2} \,\rangle$

   - $\Sigma^{\mathcal{M}_1 \cup \mathcal{M}_2} = (\Sigma^{\mathcal{M}_1} \cup \Sigma^{\mathcal{M}_2}) \setminus (\Sigma_d^{\mathcal{M}_1} \cup \Sigma_d^{\mathcal{M}_2}) = (\{add, mult\} \cup \emptyset) \setminus (\{\mathit{diff}\} \cup \{add, mult\}) = \{add, mult\} \setminus \{\mathit{diff}, add, mult\} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_1 \cup \mathcal{M}_2} = \{\mathit{diff}\} \cup \{add, mult\} = \{\mathit{diff}, add, mult\}$
   - $\mathcal{P}_{\mathcal{M}_1 \cup \mathcal{M}_2} = \{$

     $$\begin{array}{lcl} \mathit{diff}\,(\lambda u.F, x) & \to & zero \\ \mathit{diff}\,(\lambda u.u, x) & \to & s(zero) \\ \mathit{diff}\,(\lambda u.add(F(u), G(u)), x) & \to & add(\mathit{diff}\,(\lambda u.F(u), x), \mathit{diff}\,(\lambda u.G(u), x)) \\ \mathit{diff}\,(\lambda u.mult(F(u), G(u)), x) & \to & add(mult(\mathit{diff}\,(\lambda u.F(u), x), G(x)), \\ & & \quad mult(\mathit{diff}\,(\lambda u.G(u), x), F(x))) \\ \\ add(x, zero) & \to & x \\ add(x, s(y)) & \to & s(add(x, y)) \\ \\ mult(x, zero) & \to & zero \\ mult(x, s(y)) & \to & add(x, mult(x, y)) \end{array}$$

     $\}$

2. $\mathcal{M}_2 \backslash \Sigma_1 = \langle\, \Sigma^{\mathcal{M}_2 \backslash \Sigma_1}, \Sigma_d^{\mathcal{M}_2 \backslash \Sigma_1}, \mathcal{P}_{\mathcal{M}_2 \backslash \Sigma_1} \,\rangle$

   - $\Sigma^{\mathcal{M}_2 \backslash \Sigma_1} = \Sigma'^{\mathcal{M}_2} = \{add\}$
   - $\Sigma_d^{\mathcal{M}_2 \backslash \Sigma_1} = \Sigma_d^{\mathcal{M}_2} \setminus \Sigma = \{add, mult\} \setminus \{add\} = \{mult\}$
   - $\mathcal{P}_{\mathcal{M}_2 \backslash \Sigma_1} = \{$

     $$\begin{array}{lcl} mult(x, zero) & \to & zero \\ mult(x, s(y)) & \to & add(x, mult(x, y)) \end{array}$$

     $\}$

3. $\mathcal{M}_3 \ll \mathcal{M}_2 = \mathcal{M}_3 \cup (\mathcal{M}_2 \setminus \Sigma_d^{\mathcal{M}_3}) = \langle\, \Sigma^{\mathcal{M}_3 \ll \mathcal{M}_2}, \Sigma_d^{\mathcal{M}_3 \ll \mathcal{M}_2}, \mathcal{P}_{\mathcal{M}_3 \ll \mathcal{M}_2} \,\rangle$

   - $\Sigma^{\mathcal{M}_3 \ll \mathcal{M}_2} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_3 \ll \mathcal{M}_2} = \{add, mult\}$
   - $\mathcal{P}_{\mathcal{M}_3 \ll \mathcal{M}_2} = \{$

     $$\begin{array}{lcl} add(zero, x) & \to & x \\ add(s(y), zero) & \to & s(add(y, x)) \\ \\ mult(x, zero) & \to & zero \\ mult(x, s(y)) & \to & add(x, mult(x, y)) \end{array}$$

     $\}$

4. $\mathcal{M}_2[\mathcal{M}_4, \theta] = \mathcal{M}_4 \ll \theta(\mathcal{M}_2) = \mathcal{M}_4 \cup (\theta(\mathcal{M}_2) \setminus \Sigma_d^{\mathcal{M}_4}) = \langle\, \Sigma^{\mathcal{M}_2[\mathcal{M}_4,\theta]}, \Sigma_d^{\mathcal{M}_2[\mathcal{M}_4,\theta]}, \mathcal{P}_{\mathcal{M}_2[\mathcal{M}_4,\theta]} \,\rangle$

   - $\Sigma^{\mathcal{M}_2[\mathcal{M}_4,\theta]} = \emptyset$
   - $\Sigma_d^{\mathcal{M}_2[\mathcal{M}_4,\theta]} = \{addswap, mult\}$
   - $\mathcal{M}_2[\mathcal{M}_4, \theta] = \{$

$$
\begin{array}{lcl}
addswap(zero, x) & \rightarrow & x \\
addswap(s(y), zero) & \rightarrow & s(addswap(y, x)) \\
 & & \\
mult(x, zero) & \rightarrow & zero \\
mult(x, s(y)) & \rightarrow & add(x, mult(x, y))
\end{array}
$$

   $\}$

4.3.2 Compositional and Fully Abstract Semantics

An important aspect to be considered when a declarative language is extended for modular programming is the sound integration of the behavior of the modular operations into the semantics of the language. In this setting, the properties of *compositionality* and *full abstraction* have been recognized as two fundamental concepts in the studies on the semantics of declarative programming languages [13]. Simply stated, a semantics is compositional in a modular approach if semantically equivalent program modules are indistinguishable by means of module operations, that is, the meaning of a module can be obtained from the meaning of its components. Compositionality of the semantics ensures that program modules which are semantically equivalent can be replaced with other ones without affecting the intended semantics of the whole system. For instance, this property establishes a firm foundation for reasoning about programs and program transformations. Suppose that a module $\mathcal{M}$ consists of several components $\mathcal{M}_1, \ldots, \mathcal{M}_n$, suitable composed together by means of module operations. Suppose also that $\mathcal{M}_i'$ is a more efficient version of $\mathcal{M}_i$, obtained for instance by applying some program transformation technique to program $\mathcal{P}_i$ that corresponds to module $\mathcal{M}_i$. If $\mathcal{P}_i'$ (corresponding to $\mathcal{M}_i'$) is equivalent to $\mathcal{P}_i$ in the chosen semantics then the property of compositionality ensures that the substitution of $\mathcal{M}_i'$ for $\mathcal{M}_i$ will not affect the meaning of the whole module $\mathcal{M}$. On the other hand, the property of full abstraction establishes that the equivalence relation induced by the semantics is the largest equivalence relation that can be used to substitute program modules without affecting the intended semantics of the whole system. In other words, a semantics is fully abstract if indistinguishable program modules are semantically equivalent.

In this section we deal with a modular semantics for program modules thanks to the *pattern algebra transformer* defined in Section 4.2. This immediate conse-

quence operator captures directly the information concerning possible compositions obtained by the union of signatures and rules, and the corresponding semantics is directly *compositional* by construction with respect to the union operation of program modules. However, in order to obtain the complementary property of compositionality, the so-called *full abstraction* property, the adequacy of this semantics must be established with respect to the deletion operation of a signature in a program module, used to delete whole sets of program rules defining functions. We define a compositional and fully abstract semantics for the reduced set of operations on program modules defined in Section 4.3.1, union and deletion, that are enough to express the most extended ways of composing modules and their relationships.

In this work we adopt an approach inspired in [13, 74], where compositionality and full abstraction are defined in terms of the equivalence relation induced by the semantics. In the sequel we consider *CPRS*-programs instead of *CPRS*-modules since modules can be easily deduced from programs in an underlying universal signature, and this makes definitions and results easier to follow.

Definition 4.3.4 (Modular Semantics) *Let us consider a GHRC-semantic $\mathcal{S}$ and its corresponding equivalence relation $\sim^{\mathcal{S}}$ (i.e., two CPRS-programs are $\sim^{\mathcal{S}}$-equivalent if and only if they have the same meaning in the semantics $\mathcal{S}$). We define:*

1. *$\mathcal{S}$ is compositional if:*

   (a) *For all CPRS-programs $\mathcal{P}$ and $\mathcal{Q}$:*

   $$\mathcal{P} \sim^{\mathcal{S}} \mathcal{Q} \;\Rightarrow\; \mathcal{M}_{\mathcal{P}} = \mathcal{M}_{\mathcal{Q}}$$

   (b) *For all CPRS-programs $\mathcal{P}_i$ and $\mathcal{Q}_i$ and signature Σ, for all $i \in \{1, \ldots, n\}$, and all $Op \in \{\cup, (\cdot)\backslash\Sigma\}$:*

   $$\mathcal{P}_i \sim^{\mathcal{S}} \mathcal{Q}_i \;\Rightarrow\; Op(\overline{\mathcal{P}_n}) \sim^{\mathcal{S}} Op(\overline{\mathcal{Q}_n})$$

2. *$\mathcal{S}$ is fully abstract if and only if for all CPRS-programs $\mathcal{P}$ and $\mathcal{Q}$:*

   $$\mathcal{M}_{\mathtt{C}[\![\mathcal{P}]\!]} = \mathcal{M}_{\mathtt{C}[\![\mathcal{Q}]\!]} \;\Rightarrow\; \mathcal{P} \sim^{S} \mathcal{Q}$$

   for all context $\mathtt{C}$, where contexts $\mathtt{C}[\![\mathcal{X}]\!]$ are inductively defined as follows: the metavariable $\mathcal{X}$ and each CPRS-program is a context, and for each $Op \in \{\cup, (\cdot)\backslash\Sigma\}$ and $\mathtt{C}_1, \ldots, \mathtt{C}_n$ contexts, $Op(\mathtt{C}_1, \ldots, \mathtt{C}_n)$ is a context.

To find a compositional semantics in our higher-order programming framework we can build *CPRS*-programs from other *CPRS*-programs adding program rules for new functions or for already defined functions, and consider them as pattern algebra

transformers as done in [13, 74]. However, to obtain full abstraction with respect to the deletion operation, we have to consider pattern models of *CPRS*-programs obtained by deleting any signature.

Definition 4.3.5 (Transformer Semantics)

- *We define the pattern algebra transformer semantics* $\mathbb{T}$ *of a CPRS-program* $\mathcal{P}$ *by denoting the meaning of* $\mathcal{P}\backslash\Sigma$*, for all signature* Σ*, by its pattern algebra transformer* $\mathbb{T}_{\mathcal{P}\backslash\Sigma}$*. By applying Theorem 4.2.7:*

$$[\![\mathcal{P}]\!]_{\mathbb{T}} = \{\mathcal{M} \mid \mathcal{M} \text{ is a pattern model of } \mathcal{P}\backslash\Sigma\}$$

  for all signature Σ.

- *Two CPRS-programs* $\mathcal{P}$ *and* $\mathcal{Q}$ *are* $\sim^{\mathbb{T}}$*-equivalents if and only if both define the same pattern algebra transformer by deleting any signature. By applying Theorem 4.2.7:*

$$\mathcal{P} \sim^{\mathbb{T}} \mathcal{Q} \;\Leftrightarrow\; [\![\mathcal{P}]\!]_{\mathbb{T}} = [\![\mathcal{Q}]\!]_{\mathbb{T}}$$

We are ready to state and prove the properties of compositionality and full abstraction of the pattern algebra transformer semantics $\mathbb{T}$, which justify the adoption of this semantics as the main point of this chapter.

Theorem 4.3.6 (Modularity of $\mathbb{T}$) *The pattern algebra transformer semantics* $\mathbb{T}$ *is compositional and fully abstract with respect to the set of operations* $\{\cup, (\cdot)\backslash\Sigma\}$.

Proof We prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the set of operations $\{\cup, (\cdot)\backslash\Sigma\}$. First, from $\mathcal{P} \sim^{\mathbb{T}} \mathcal{Q}$ we deduce that $\mathcal{P}\backslash\Sigma$ and $\mathcal{Q}\backslash\Sigma$ have the same pattern models for all signature Σ. In particular, for the empty signature, $\mathcal{P}$ and $\mathcal{Q}$ have the same least pattern models and the same least fixed-points. Thus, we conclude that $\mathcal{M}_{\mathcal{P}} = \mathcal{M}_{\mathcal{Q}}$. Now, we prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the union of programs: $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$, for $i = 1, ..., n$, implies $\bigcup_{i=1}^n \mathcal{P}_i \sim^{\mathbb{T}} \bigcup_{i=1}^n \mathcal{Q}_i$. Since $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$, both define the same pattern algebra transformer $\mathbb{T}_{\mathcal{P}_i\backslash\Sigma} = \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}$ for all signature Σ. Let $\mathcal{M}$ be a pattern model of $\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma$. From Theorem 4.2.7: $\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{P}_i\backslash\Sigma}(\mathcal{M}) = (\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{P}_i\backslash\Sigma})(\mathcal{M}) = \mathbb{T}_{\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$. Therefore, $\mathbb{T}_{\mathcal{P}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for $i = 1, \ldots, n$. Moreover, from $\mathcal{P}_i \sim^{\mathbb{T}} \mathcal{Q}_i$ we obtain $\mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$ for $i = 1, \ldots, n$, and then $\mathbb{T}_{\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) = (\bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma})(\mathcal{M}) = \bigsqcup_{i=1}^n \mathbb{T}_{\mathcal{Q}_i\backslash\Sigma}(\mathcal{M}) \sqsubseteq \mathcal{M}$. Therefore, $\mathcal{M}$ is a pattern model of $\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma$. By reasoning in a similar way, it can be obtained that all pattern models of $\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma$ are also pattern models of $\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma$, and this proves that $\mathbb{T}_{\bigcup_{i=1}^n \mathcal{P}_i\backslash\Sigma} = \mathbb{T}_{\bigcup_{i=1}^n \mathcal{Q}_i\backslash\Sigma}$. Finally, since $(\bigcup_{i=1}^n \mathcal{P}_i)\backslash\Sigma = \bigcup_{i=1}^n (\mathcal{P}_i\backslash\Sigma)$, we

deduce that $\mathbb{T}_{(\bigcup_{i=1}^{n} \mathcal{P}_i)\backslash\Sigma} = \mathbb{T}_{(\bigcup_{i=1}^{n} \mathcal{Q}_i)\backslash\Sigma}$ for all signature Σ, which means $\bigcup_{i=1}^{n} \mathcal{P}_i \sim^{\mathbb{T}} \bigcup_{i=1}^{n} \mathcal{Q}_i$.

Second, we prove that the pattern algebra transformer semantics $\mathbb{T}$ is compositional with respect to the deletion of a signature in a program: $P \sim^{\mathbb{T}} Q$ implies $\mathcal{P}\backslash\Sigma' \sim^{\mathbb{T}} \mathcal{Q}\backslash\Sigma'$, for every signature Σ'. Since $P \sim^{\mathbb{T}} Q$, both define the same pattern algebra transformer $\mathbb{T}_{\mathcal{P}\backslash(\Sigma\cup\Sigma')} = \mathbb{T}_{\mathcal{Q}\backslash(\Sigma\cup\Sigma')}$ for all signatures Σ and Σ'. Since $\mathcal{R}\backslash(\Sigma \cup \Sigma') = (\mathcal{R}\backslash\Sigma)\backslash\Sigma'$ for every *CPRS*-program $\mathcal{R}$, we deduce that $\mathbb{T}_{(\mathcal{P}\backslash\Sigma)\backslash\Sigma'} = \mathbb{T}_{(\mathcal{Q}\backslash\Sigma)\backslash\Sigma'}$ for every signatures Σ and Σ', which means that $\mathcal{P}\backslash\Sigma' \sim^{\mathbb{T}} \mathcal{Q}\backslash\Sigma'$, for every signature Σ'. Finally, to prove that the pattern algebra transformer semantics $\mathbb{T}$ is fully abstract, we only need to prove that $\mathcal{P} \nsim^{\mathbb{T}} \mathcal{Q}$ implies that there exists a context $\mathtt{C}$ where we can discriminate the observable behavior of both programs. From $\mathcal{P} \nsim^{\mathbb{T}} \mathcal{Q}$ we deduce that there exists a signature Σ such that the pattern algebra transformers $\mathbb{T}_{\mathcal{P}\backslash\Sigma}$ and $\mathbb{T}_{\mathcal{Q}\backslash\Sigma}$ are different. Then there exists a context $\mathtt{C}'$ such that $\mathcal{M}_{\mathtt{C}'[\![\mathcal{P}\backslash\Sigma]\!]} \neq \mathcal{M}_{\mathtt{C}'[\![\mathcal{Q}\backslash\Sigma]\!]}$. Thus by considering the new context $\mathtt{C}[\![\mathcal{R}]\!] = \mathtt{C}'[\![\mathcal{R}\backslash\Sigma]\!]$ we have that $\mathcal{M}_{\mathtt{C}[\![\mathcal{P}]\!]} \neq \mathcal{M}_{\mathtt{C}[\![\mathcal{Q}]\!]}$.

□

The modularity of the $\mathbb{T}$ semantics is particularly relevant in declarative programming, because one of the most critical aspects in multi-paradigms declarative systems is the possibility of making a separate compilation of modules, and this can only be made in the presence of some kind of compositionality. For instance, $\mathcal{TOY}$ is a constraint functional logic system, designed to support the main declarative programming styles and their combination. The current version provides a module system involving higher-order patterns, a polymorphic type system, constraints with symbolic equations and disequations, linear and non-linear arithmetic constraints over real numbers, and finite domain constraints. For this reason, our modular semantics for higher-order declarative constraint programming can be applied to the $\mathcal{TOY}$ system to allow each distinct module to be compiled separately, with the effect of inlining being realized by a subsequent linking process. Moreover, modular development installs boundaries in programs that can be important to the practical use of static analysis techniques and that are fundamental to the notion of separate compilation and testing.

Chapter 5

Conclusions and Future Work

In this work we have described a new semantic framework with λ-abstractions for functional logic programming. Now we describe the main conclusions from each chapter of the work:

In Chapter 2 we have presented the syntax and basic concepts about the higher-order functional logic programming language with λ-abstractions that is the object of study of this work. We first have described two classical and well-known frameworks that are at the basis of our work: term rewriting systems and λ-calculus. Finally in that chapter, we have described in detail the language we have studied. We have adapted and completed some of the definitions given in the previous sections, that have been useful when defining logics and semantics for the programming framework in the following chapters.

In Chapter 3 we have presented a rewriting logic for the higher-order programming language with λ-abstractions introduced in Chapter 2. This logic defines the reduction relation among terms for a given *CPRS* in a natural way that is also useful to reason about programs and has been adapted from [23] and [24]. In the first section of this chapter we have discussed the problem of higher-order unification that motivates the apparently arbitrary election of patterns in Chapter 2. In the second section we have described the rewriting logic *GHRC* and their associated proof calculus. In the last section of the chapter, we have presented some properties about the proof calculus associated to this logic that are essential to prove basic results about the declarative semantics of the higher-order functional logic programming language with λ-abstractions in Chapter 4.

In Chapter 4 we have presented the declarative foundations of our semantic framework with λ-abstractions by means of a model-theoretic and a fixed-point semantics, proving basic results for each of them. Model-theoretic semantics denote

models of *CPRS* programs as algebras that satisfy the pattern rewrite rules of the program. Fixed-point semantics characterizes the pattern model as the least fixed-point of an operator on algebras, that coincides with the idea of successive refinements. Finally we have defined *CPRS* program modules and have proved that the modular semantics defined for the framework is compositional and fully abstract with respect to the defined operations on modules, which is essential to support program transformations, modular verification, and advanced constraint-solving extensions based on this framework. This extension to the framework to support a module system has been accepted for publication in [27].

In Appendix A we have presented a higher-order unification algorithm on patterns and we have proved its main properties, soundness and completeness. This algorithm, at an early stage of development, is included for the sake of completity of the material presented in Chapter 3 and is adapted from a strict equality constraint solver of λ-equations proposed for cooperation of generic algebraic constraint domains in [25].

5.1 Contributions

Now we present the main contributions of this work:

- We have presented a revised and extended version of the semantic framework for higher-order functional logic programming language with λ-abstractions that has been developed in [23, 24] in the context of different research topics at the 'Declarative Programming Group' [23, 24, 25, 26]. We have concreted the formal theoretical basis and have compiled the most important semantic results (Theorem 4.1.8 and Theorem 4.2.7) that have been obtained for it and have presented them in a natural and more detailed way that it was done before in [23, 24].
- We have completed the framework with a modular semantics that opens the possibility to more advanced research (as explained in the next section where we describe the future work). Also, the extension is useful on its own, establishing sound theoretical foundations for any modular system for any language that can use this framework as a model of computation (Theorem 4.3.6).
- We have developed our framework from its original theoretical foundations, that are term rewriting systems and λ-calculus. We have established the sources of any notion we use that comes from those two and clarified the criterion followed on notions influenced by both of them. This contributes to clarify concepts and serves as explanation of technical conditions that appear in more advanced features of our framework, such as the definition of higher-order patterns (Definition 2.3.6) and values (Definition 3.1.3).

- We have presented a higher-order unification algorithm in Appendix A that is at an early development stage but is proved to be correct and complete with respect to the *GHRC* logic. Adapted versions of this unification algorithm can be a key feature of operational semantics based on the framework and our algorithm constitutes a good starting point for developing any of them. This idea has been already applied in the context of constraint domains to develop an operational semantics for higher-order equations solving [25].
- This framework has been used in formal verification, algorithmic debugging and higher-order constraint domains [24, 25, 26]. That is because we present a flexible and extensible approach that have applications in many related fields, thanks to the solid theoretical foundations and the expressiveness of the corresponding semantics that we define.

5.2 Future Work

The approach we employ in this work has been proven to be adequate to develop extensions based on this framework. More precisely, we have defined the following lines of work that we plan to follow in the future:

- This framework is based on patterns defined as simply-typed λ-terms; even though this is acceptable and the language constitutes a Turing-complete model of computation, it would be nice to extend it to polymorphic types to increase its expressivity. Even though higher-order patterns are problematic in the context of types [40], recent results in this area [59] may be considered to extend the framework to deal with polymorphic types at its foundations.
- Unification algorithms have been a matter of research since the earliest studies about the logic programming paradigm [86] and an algorithm for a high expressive language as ours deserves attention on its own. We think it would be interesting to provide a suitable implementation of it and compare it to another algorithms developed for similar languages such as [83] and study in depth its behavior and efficiency.
- The current implementation of the $\mathcal{TOY}$ system [62], also developed in the of context the 'Declarative Programming Group' in the last decades, is not able currently to deal with λ-abstractions and higher-order unification, even though it presents some higher-order features based on the framework described in [39]. We think that it would be interesting to incorporate our higher-order unification algorithm and the capacity to support λ-abstractions to the $\mathcal{TOY}$ system.
- An efficient implementation of the unification algorithm presented in Appendix A is very difficult to achieve due to the indeterminism source that introduces

the application of several projection rules. We think that an improved version of the algorithm oriented to the design of an efficient operational semantics could lead to improvements to the newest one developed for the framework presented in [23].

- Following the ideas from [20], it is possible to represent the instantiation of our proof calculus *GHRC* (see Section 3.2) to a particular *CPRS*-program as an equivalent *higher-order logic theory* suitable to the *Isabelle theorem prover* [78]. With this theory, it is possible to certify the validity of *GHRC*-proofs within this theorem prover, but also to prove more complex properties about programs represented in *Higher-Order Logic* (*HOL*). The *HOL*-theory obtained by this procedure would allow not only to generate certificates for reduction and equality statements, but also for more expressive and interesting properties on declarative programs. It is a matter of undoubtable interest the formalization of a translation from *CPRS*-programs to equivalent higher-order logic theories, and the efficient implementation of such a transformation; also, it would be very interesting to have access to automated methods to generate certificates in an automatic or semiautomatic way of properties about particular *CPRS*-programs.

- Extension of the set of operations defined with modules in Section 4.3 to support more expressive ones and improve the practical applicability of our modular system to higher-order functional logic programming languages. For example, it would be useful to have operations for making visible or hiding some defined function definitions or signatures of modules and specific import and export operations, following the lines of [12, 13].

- The modular semantics we present in Section 4.3 has been extended to a context with constraint domains in our recent publication [27]. Program modules are of special interest in that context because they allow to represent simpler pure constraint domains $\mathcal{D}$ as modules and an straightforward representation of *hybrid constraint domains* $\mathcal{C}$ [29] by means of operations between modules. This modular design supports the cooperation and coordination of predefined $\mathcal{D}$-modules, so more declarative and efficient solutions for practical problems can be promoted. Figure 5.1 illustrates the modular design of the $\mathcal{TOY}$ system for the cooperation among predefined $\mathcal{D}$-modules, and the mechanisms for communication and coordination via bridges, projections, functional variable applications, interpolations, and some more ad hoc operations [25]. The next example, that uses some specific notation from constraint domains, illustrates the use of modular design in this context:

  Example 5.2.1 (Modular Constraint Cooperation [25]) *A common problem in engineering is the approximation of a complicated continuous function*

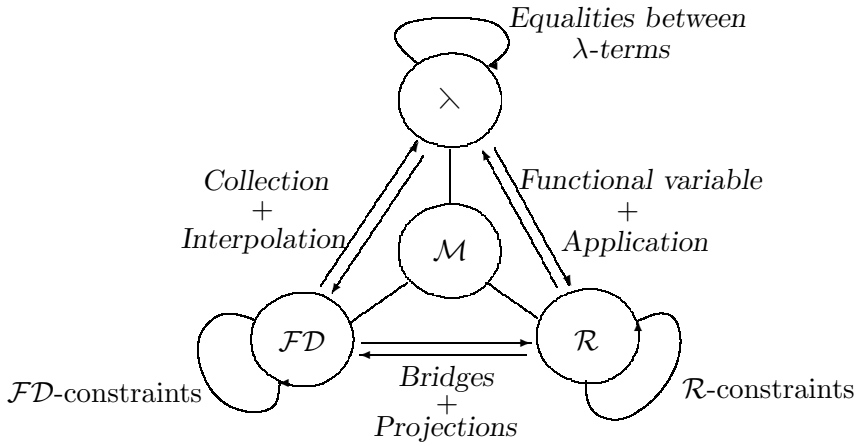


Figure 5.1: Modular design of the higher-order coordination constraint domain $\mathcal{C}$.

by a simple discrete function (e.g., the approximation of GPS satellite coordinates). Suppose we know a real function (given by a λ-abstraction $\lambda u.\, F(u)$) but it is too complex to evaluate efficiently. Then we could pick a few approximated (integer) data points from the complicated function, and try to interpolate those data points to construct a simpler function, for example, a polynomial $\lambda u.\, P(u)$. We propose the following interface of a $\mathcal{C}$-module to approximate a continuous function represented by a lambda abstraction $\lambda u.\, F(u)$ over real numbers by a discrete polynomial function $\lambda u.\, P(u)$ over integer numbers; notice that we need an additional component for modules that is the first one in the example and represents domain specific functions:

$$
\begin{array}{lcl}
GPS & = & \langle\, \{domain :: [int] \to int \to int \to bool, \\
 & & \quad labeling :: [int] \to bool, \\
 & & \quad \rightleftharpoons :: int \to real \to bool, \\
 & & \quad - :: real \to real \to real, \\
 & & \quad < :: real \to real \to bool, \\
 & & \quad |\cdot| :: real \to real\}, \\
 & & \{collection :: int \to int \to [(int, int)] \to bool, \\
 & & \quad interpolation :: [(int, int)] \to (int \to int) \to bool\}, \\
 & & \{disc :: (real \to real) \to (int \to int)\}\, \rangle
\end{array}
$$

where the corresponding body of the module is:

$$
\begin{array}{l}
disc\,(\lambda u.\, F(u)) \to \lambda u.\, P(u) \Leftarrow \\
\qquad\qquad\qquad\qquad domain\ [X]\ 0\ N,\ labeling\ [X], \\
\qquad\qquad\qquad\qquad X \rightleftharpoons RX,\ Y \rightleftharpoons RY, \\
\qquad\qquad\qquad\qquad |\, F(RX) - RY\, | < 1, \\
\qquad\qquad\qquad\qquad collection\ X\ Y\ C,\ interpolation\ C\ P
\end{array}
$$

The $\mathcal{C}$-module GPS *uses the predefined $\mathcal{FD}$-module to provide $\mathcal{FD}$-constraints*

domain $[X]$ 0 N*, labeling* $[X]$ *to generate each value of the discrete interval* $[0..N]$*. In order to model the cooperation and communication between* $\mathcal{FD}$*-modules and* $\mathcal{R}$*-modules we use a special kind of hybrid constraints* $\rightleftharpoons$ *called* bridges *provided by a predefined mediatorial domain* $\mathcal{M}$*, as a key tool for communicating constraints between different higher-order numeric constraint domains [29, 25]. More precisely, the first bridge constraint* $X \rightleftharpoons RX$ *maps each integer value of* X *into an equivalent real value in* RX*. By applying the higher-order functional variable* F *to* RX *we obtain the* $\mathcal{R}$*-constraint* $|\, F(RX) - RY \,| < 1$ *provided by the primitive functions of the predefined* $\mathcal{R}$*-module. From this constraint, the* $\mathcal{R}$*-solver computes (infinitely many) real values for* RY*. However, because of the second bridge constraint* $Y \rightleftharpoons RY$*, each real value assigned to* RY *by the constraint solving process causes the variable* Y *to be bound only to an equivalent integer value. By means of the primitive constraint collection* $X\ Y\ C$ *provided by the predefined* $\rightarrow$*-module we can collect all the pairs* (X, Y) *generated by the labeling-solving process in a set* C*. Finally, interpolation* $C\ P$ *finds a polynomial which goes exactly through the points collected in* C *by means of the* Lagrange Interpolation *method. For instance, we can consider the following goal* $disc\,(\lambda u.\, 4 * u - u^2) == \lambda u.\, P(u)$ *involving the continuous function* F *as* $\lambda u.\, 4 * u - u^2$ *with* $N = 4$*. We obtain the set of integer pairs* (x_i, y_i) *in* $C = \{(0,0), (1,3), (2,4), (3,3), (4,0)\}$*. For this particular case, it is easy to check that this computed answer is simply* $\{P \mapsto \lambda u.\, 4 * u - u^2\}$*.*

The theoretical foundations that constitutes our modular semantics open the field to the development of a *language* of high level of abstraction for constraint cooperation and coordination based on module operations following [75, 76].

Appendices

Appendix A

Unification of Higher-Order Patterns

In this appendix we present a higher-order unification algorithm on patterns and we prove its main properties, soundness and completeness. This algorithm is adapted from a strict equality constraint solver of λ-equations proposed for cooperation of generic algebraic constraint domains in [25].

A.1 The Higher-Order Unification Algorithm

In this section we present the higher-order unification algorithm by means of some transformation rules over *states*, that consists on a set of equations to solve and a set of patterns that represents the result of the unification process.

Definition A.1.1 (States) *The unification algorithm on patterns acts on* **states** *of the form $P \equiv \langle E \,|\, \mathcal{K} \rangle$, where E is a set of equations $s \approx t$ interpreted as strict equalities between λ-terms s, t, and $\mathcal{K}$ is a set of patterns intended to represent and store computed values during the unification process.*

Solving an unification problem proceeds by means of the application of successive transformation steps on states in the form of a derivation. In the transformation rules for the unification algorithm, equations $s \approx t$ are interpreted as strict equalities $s == t$.

Definition A.1.2 (Higher-Order Unification Algorithm)

- *An* **unification-derivation** *of a set E of strict equations between λ-terms is a maximal finite sequence of transformation steps: $P_0 \equiv \langle E \,|\, \emptyset \rangle \equiv \langle E_0 \,|\, \mathcal{K}_0 \rangle \Rightarrow_{\sigma_1} P_1 \equiv \langle E_1 \,|\, \mathcal{K}_1 \rangle \Rightarrow_{\sigma_2} \cdots \Rightarrow_{\sigma_m} P_m \equiv \langle E_m \,|\, \mathcal{K}_m \rangle$, between states $P_0, P_1, \ldots, P_m$, such that $P_m \neq$ `fail` is a final state, i.e., a non-failure state which cannot be transformed anymore.*

- *Each transformation step in an unification-derivation* Π *corresponds to an instance of some transformation rule described below. We abbreviate* Π *by* $P_0 \Rightarrow_\sigma^* P_m$*, where* $\sigma = \sigma_1 \cdots \sigma_m$.

(an) **annotation**
$\langle \{\!\{ s \approx t, E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_{\{\}} \langle \{\!\{ s \approx_H t, E \}\!\} \mid \mathcal{K} \cup \{H\} \rangle$
where H is a fresh variable of a suitable type.

(sg) **strict guess**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) \approx_H t, E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) \approx_{H\sigma} t, E \}\!\} \mid \mathcal{K}\sigma \rangle$
where $a \in \mathcal{F} \cup \{\overline{x_k}\}$, and $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$.

(d) **decomposition**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) \approx_u \lambda\overline{x_k}.a(\overline{t_n}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ \overline{\lambda\overline{x_k}.s_n \approx_{H_n} \lambda\overline{x_k}.t_n}, E \}\!\} \mid \mathcal{K}\sigma \rangle$
where $a \in \mathcal{F} \cup \{\overline{x_k}\}$, and either

- $u \equiv H$ and $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$, or
- $u \equiv \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})$ and $\sigma = \varepsilon$.

(i) **imitation**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) \approx_u \lambda\overline{x_k}.f(\overline{t_n}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ \overline{\lambda\overline{x_k}.X_n(\overline{s_p}) \approx_{H_n} \lambda\overline{x_k}.t_n}, E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma \rangle$
where $X \in \mathcal{V}$, and either

- $u \equiv H$ and $\sigma = \{X \mapsto \lambda\overline{y_p}.f(\overline{X_n(\overline{y_p})}), H \mapsto \lambda\overline{x_k}.f(\overline{H_n(\overline{x_k})})\}$, or
- $u \equiv \lambda\overline{x_k}.f(\overline{H_n(\overline{x_k})})$ and $\sigma = \{X \mapsto \lambda\overline{y_p}.f(\overline{X_n(\overline{y_p})})\}$.

(p) **projection**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) \approx_u t, E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) \approx_u t, E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma \rangle$
where $X \in \mathcal{V}$, t is not flex, and $\sigma = \{X \mapsto \lambda\overline{y_p}.y_i(\overline{X_n(\overline{y_p})})\}$.

(fs) **flex same**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{y_p}) \approx_H \lambda\overline{x_k}.X(\overline{y'_p}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma \rangle$
where $X \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_p})$ and $\lambda\overline{x_k}.X(\overline{y'_p})$ are patterns, $\sigma = \{X \mapsto \lambda\overline{y_p}.Z(\overline{z_q}), H \mapsto \lambda\overline{x_k}.Z(\overline{z_q})\}$ with $\{\overline{z_q}\} = \{y_i \mid y_i = y'_i,\ 1 \leq i \leq n\}$.

(fd) **flex different**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{y_p}) \approx_H \lambda\overline{x_k}.Y(\overline{y'_q}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{ E \}\!\}\sigma \mid (\mathcal{K} \cup \{X, Y\})\sigma \rangle$
where $X, Y \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_p})$ and $\lambda\overline{x_k}.Y(\overline{y'_q})$ are patterns, $X \neq Y$, $\sigma = \{X \mapsto \lambda\overline{y_p}.Z(\overline{z_r}), Y \mapsto \lambda\overline{y'_q}.Z(\overline{z_r}), H \mapsto \lambda\overline{x_k}.Z(\overline{z_r})\}$ with $\{\overline{z_r}\} = \{\overline{y_p}\} \cap \{\overline{y'_q}\}$.

(cf) **clash failure**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) \approx_u \lambda\overline{x_k}.a'(\overline{t_m}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_{\{\}} \texttt{fail}$
if $a, a' \in \Sigma_c \cup \{\overline{x_k}\}$, and either (i) $a \neq a'$ or (ii) $hd(u) \notin \mathcal{V} \cup \{a, a'\}$.

(oc) **occur check**
$\langle \{\!\{ \lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.X(\overline{y_n}), E \}\!\} \mid \mathcal{K} \rangle \Rightarrow_{\{\}} \texttt{fail}$
if $X \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_n})$ is a flex pattern, $hd(\lambda\overline{x_k}.s) \neq X$ and $(\lambda\overline{x_k}.s)|_p = X(\overline{z_n})$, where $\overline{z_n}$ is a sequence of distinct bound variables and p is a safe position of $\lambda\overline{x_k}.s$.

In the sequel, we will describe the main properties of the unification algorithm according to a suitable semantics for states and unification-derivations. The general idea is to ensure the computation of solutions from a unification problem which are correct with respect to the semantics given by the *GHRC* logic defined in Section 3.2.

Definition A.1.3 (Meaning of States and Computed Answers)

- *The* ***meaning of a state*** $P \equiv \langle E \,|\, \mathcal{K} \rangle$ *is as follows:* $[\![\langle E \,|\, \mathcal{K} \rangle]\!] = \{\gamma \in Soln(E) \mid \mathcal{K}\gamma$ *is a set of values*$\}$. *We note that* $[\![\langle E \,|\, \mathcal{K} \rangle]\!] = \emptyset$ *whenever* $\mathcal{K}$ *is not a set of values. In the sequel, we denote this state by* `fail` *and call it failure state.*

- *Given a set* E *of* λ*-equations, the set of* ***computed answers*** *produced by the unification algorithm is* $\mathcal{A}(E) = \{\, \sigma\gamma|_{\mathcal{FV}(E)} \,|\, \langle E \,|\, \emptyset \rangle \Rightarrow_\sigma^* P$ *is an unification-derivation and* $\gamma \in [\![P]\!] \,\}$.

Since the design considerations are quite involved and the analysis techniques quite complicated, we consider it useful to precede the presentation of the main properties of the unification algorithm with a brief outline of our design considerations and techniques.

Typical requirements in the design of such an algorithm based on transformation steps are *soundness* (every computed answer is a solution, i.e., $\mathcal{A}(E) \subseteq Soln(E)$), and *completeness* (for any $\gamma \in Soln(E)$ there exists $\gamma' \in \mathcal{A}(E)$ such that $\gamma' \leq \gamma\ [\mathcal{FV}(E)]$). Note that the completeness requirement demands the capability to compute a *minimal complete* set of solutions. It is easy to see that if the higher-order unification algorithm is complete then it suffices to enumerate minimal complete set of solutions of the final states. Therefore, an important design issue is to guarantee that minimal complete sets of solutions are easy to read off for the final states. In the design of first-order unification algorithms, this is achieved by ensuring that final states have empty components; thus the minimal complete set of solutions of a final state consists of the identity substitution ε. Unfortunately, things are much more complicated in the higher-order case. This problem is inevitably related to the problem of unifying *flex* λ*-terms* (i.e., λ-terms t such that $hd(t) \in \mathcal{FV}(t)$), which is in general intractable. We adopt an approach similar to *Huèt's procedure of higher-order pre-unification* [70, 81]. We refrain from solving equations between flex λ-terms as much as possible. As a consequence, our final states will be a class of states whose set of λ-equations are only between flex λ-terms. This guarantees that the final states are meaningful and that it is relatively easy to read off some of their solutions.

A.2 Soundness and Completeness of the Algorithm

The main properties of the unification algorithm, *soundness* and *completeness*, relate the solutions of a set of equations to the answers computed by our system of transformation rules for higher-order unification.

Lemma A.2.1 (Local soundness of a single unification step) *If $P \Rightarrow_\sigma P'$ is an unification step then $\{\sigma\gamma \mid \gamma \in [\![P']\!]\} \subseteq [\![P]\!]$. Moreover, if P satisfies the preconditions of a transformation rule for failure detection, then $[\![P]\!] = \emptyset$.*

Proof Let $\varphi : P \Rightarrow_\sigma P'$ be a transformation step, where $P = \langle \{\!\{C, E\}\!\} \mid \mathcal{K}\rangle$ with C the selected equation, $P' = \langle \{\!\{C', E\sigma\}\!\} \mid \mathcal{K}'\rangle$, C' is the (possibly empty) sequence of descendants of C in P', and $\gamma \in [\![P']\!]$. First, we note that the unification algorithm is defined in such a way that $\mathcal{K}\sigma \subseteq \mathcal{K}'$. Therefore, $\mathcal{K}\sigma\gamma \subseteq \mathcal{K}'\gamma$, which is a set of values because $\gamma \in [\![\langle \{\!\{C', E\sigma\}\!\} \mid \mathcal{K}'\rangle]\!]$. We also note that if P is a state then $\sigma\gamma \in \mathit{Soln}(E)$ whenever $\gamma \in \mathit{Soln}(E\sigma)$. Thus, we only have to show $\sigma\gamma \in \mathit{Soln}(C)$. Our proof is by case distinction on the transformation rule of the unification algorithm employed in φ:

- If φ is an (an)-step $\langle \{\!\{s \approx t, E\}\!\} \mid \mathcal{K}\rangle \Rightarrow_{\{\}} \langle \{\!\{s \approx_H t, E\}\!\} \mid \mathcal{K} \cup \{H\}\rangle$ then $\sigma = \varepsilon$ and $\sigma\gamma = \gamma$. Lemma A.2.1 holds in this case because $\mathit{Soln}(s \approx_H t) \subseteq \mathit{Soln}(s \approx t)$.

- Suppose φ is a (sg)-step which selects the equation $\lambda\overline{x_k}.a(\overline{s_n}) \approx_H t$ with $a \in \{\overline{x_k}\} \cup \Sigma$ and yields the descendant $\lambda\overline{x_k}.a(\overline{s_n}) \approx_{H\sigma} t$ with $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$. Then $H\sigma\gamma$ is a total λ-term and $\sigma\gamma \in \mathit{Soln}(\lambda\overline{x_k}.a(\overline{s_n}) \approx_H t)$.

- If φ is a (d)- or an (i)- step of the form

$$\langle \{\!\{\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.t, E\}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{\overline{\lambda\overline{x_k}.s_n \approx_{H_n} \lambda\overline{x_k}.t_n}, E\}\!\}\sigma \mid \mathcal{K}'\rangle$$

  then there exists $a \in \Sigma \cup \{\overline{x_k}\}$ such that $\lambda\overline{x_k}.s\sigma = \lambda\overline{x_k}.a(\overline{s_n})$, $\lambda\overline{x_k}.t\sigma = \lambda\overline{x_k}.a(\overline{t_n})$ and $u\sigma = \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})$. To prove $\sigma\gamma \in \mathit{Soln}(\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.t)$ we use proof trees: $\mathcal{P}_i \in \mathcal{PT}^{H_i\gamma}(\lambda\overline{x_k}.s_i\sigma\gamma \approx \lambda\overline{x_k}.t_i\sigma\gamma)$, $i = 1, \ldots, n$. Then, $\mathcal{P}_i \equiv \frac{\mathcal{P}_{i,1} \quad \mathcal{P}_{i,2}}{\lambda\overline{x_k}.s_i\sigma\gamma \approx \lambda\overline{x_k}.t_i\sigma\gamma}$, where $\mathcal{P}_{i,1} \in \mathcal{PT}(\lambda\overline{x_k}.s_i\sigma\gamma \approx H_i\gamma)$ and $\mathcal{P}_{i,2} \in \mathcal{PT}(\lambda\overline{x_k}.\, t_i\sigma\gamma \approx H_i\gamma)$. Let $\mathcal{P}'_1 = \frac{\mathcal{P}_{1,1} \ \ldots \ \mathcal{P}_{n,1}}{(\lambda\overline{x_k}.s \approx u)\sigma\gamma}$ and $\mathcal{P}'_2 = \frac{\mathcal{P}_{1,2} \ \ldots \ \mathcal{P}_{n,2}}{(\lambda\overline{x_k}.t \approx u)\sigma\gamma}$. Then, $\frac{\mathcal{P}'_1 \quad \mathcal{P}'_2}{(\lambda\overline{x_k}.s \approx \lambda\overline{x_k}.t)\sigma\gamma} \in \mathit{Wtn}_{\sigma\gamma}(\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.t)$ (i.e., the set of *witnesses* that $\sigma\gamma$ is a solution of $\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.t$) because $u\sigma\gamma$ is a total λ-term. Then, $\sigma\gamma \in \mathit{Soln}(\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.t)$.

- If φ is a failure detection step then $\{\sigma\gamma \mid \gamma \in [\![\langle E' \mid \mathcal{K}'\rangle]\!]\} = \{\sigma\gamma \mid \gamma \in [\![\mathtt{fail}]\!]\} = \emptyset \subseteq [\![\langle E \mid \mathcal{K}\rangle]\!]$.

The other cases can be proved similarly. Let $P = \langle E \mid \mathcal{K}\rangle$. Suppose by contrary that P satisfies the preconditions of a transformation rule for failure detection and that there exists $\gamma \in [\![P]\!]$. If P satisfies the preconditions of (cf) then E contains $\lambda\overline{x_k}.a(\overline{s_n}) \approx_u \lambda\overline{x_k}.a'(t_m)$ with $a, a' \in \Sigma_c \cup \{\overline{x_k}\}$ and either (i) $a \neq a'$ or (ii) $hd(u) \notin \mathcal{FV} \cup \{a, a'\}$. From $\gamma \in \mathit{Soln}(E)$ results that $u\gamma$ is a total λ-term, $\lambda\overline{x_k}.a(\overline{s_n\gamma}) \approx$

$u\gamma$ and $\lambda\overline{x_k}.a'(\overline{t_m\gamma}) \approx u\gamma$. Since $a, a' \in \Sigma_c \cup \{\overline{x_k}\}$ and $hd(u\gamma) \neq \bot$, we must have $a = hd(u\gamma) = a'$, contradiction. Thus $[\![P]\!]$ must be $\emptyset$ if the preconditions of (cf) hold. If E contains an equation $\lambda\overline{x_k}.s \approx_u \lambda\overline{x_k}.X(\overline{y_n})$ which satisfies the preconditions of (oc) then $X \in \mathcal{V}$ and therefore $X\gamma$ is a total λ-term. This implies that $\lambda\overline{x_k}.X(\overline{y_n})\gamma$ is a total λ-term. Since $\lambda\overline{x_k}.X(\overline{y_n})\gamma \approx u\gamma$ and $u\gamma$ is a total λ-term, we have $\lambda\overline{x_k}.X(\overline{y_n})\gamma = u\gamma$. From the position $p \in Pos(\lambda\overline{x_k}.s)$ we get $p \in Pos(\lambda\overline{x_k}.s\gamma)$. From $\lambda\overline{x_k}.s\gamma \approx u\gamma$ we obtain that p is a maximal safe position of $u\gamma$ and $(\lambda\overline{x_k}.s\gamma)\restriction_p \approx u\gamma\restriction_p$. But $u\gamma\restriction_p$ is a total λ-term because $u\gamma$ is a total λ-term, and we get $(\lambda\overline{x_k}.s\gamma)\restriction_p = u\gamma\restriction_p$. Therefore:

$$|u\gamma\restriction_p| = |(\lambda\overline{x_k}.(s\gamma)\restriction_p)| = |X(\overline{z_n})\gamma| = |X(\overline{y_n})\gamma| = |u\gamma|. \tag{A.1}$$

We note that the assumptions of (oc) imply $p > 1^k$. Then, p is below the root position of $u\gamma$, and therefore $|u\gamma\restriction_p| < |u\gamma|$, which contradicts relation (A.1). Hence, $[\![\mathcal{P}]\!] = \emptyset$.

□

Completeness of the unification algorithm is much more difficult to ensure: we must verify that *any* solution γ of a given set of strict equations E will be eventually *approximated*, i.e., that we will eventually reach a state $\langle E' \mid \mathcal{K}'\rangle$ in the set of *admissible states* `Adm` [23] representing a computed answer γ' such that $\gamma' \leq \gamma\ [\mathcal{FV}(E)]$. This approximation process must take into account all the possible shapes of elementary goals, and make sure that *progress* can be made towards reaching $\langle E' \mid \mathcal{K}'\rangle$. We achieve this by looking at the syntactic structure of strict equations, the solution γ which we want to approximate, and the witness that γ is a solution of the given set of equations, and show how these grouped structures or triples (called *configurations* of a set of configurations `Cfg`) can be looked up for computing a representation of an approximation of γ by means of a *well-founded ordering* $\succ$ over `Cfg`. More precisely, the following definition is used to capture the structures which are reduced by the transformation steps of the unification algorithm.

Definition A.2.2 (Configuration) *A **configuration** is a triple $S = \langle P, \gamma, \psi\rangle$, with $P = \langle E \mid \mathcal{K}\rangle \in$ `Adm`, $\gamma \in [\![P]\!]$, and ψ a mapping $E' \in E \mapsto \psi(E') \in Wtn_\gamma(E')$ which associates to each equation $E' \in E$ a witness that γ is a solution of E'. S is a non-final configuration if P is a non-final state. We denote the set of admissible configurations by* `Cfg`.

The following notions are instrumental in explaining the structure reduction process of the unification algorithm. The *size* $|E|$ of an equation E is

$$|E| = \begin{cases} |s| + |t| & \text{if } E \equiv s \approx t \text{ or } E \equiv s \approx_u t, \\ \{\!\{|E_1|, \ldots, |E_n|\}\!\} & \text{if } E \equiv \{\!\{E_1, \ldots, E_n\}\!\}. \end{cases}$$

For $\gamma \in Soln(E)$, we define

$$\|E\|_\gamma = \begin{cases} |s\gamma| + |t\gamma| & \text{if } E \equiv s \approx t \text{ or } E \equiv s \approx_u t, \\ \{\!\{\|E_1\|_\gamma, \ldots, \|E_n\|_\gamma\}\!\} & \text{if } E \equiv \{\!\{E_1, \ldots, E_n\}\!\}. \end{cases}$$

We are ready now to explain what is the unification algorithm supposed to reduce. We do this via a lexicographic combination of five terminating ordering on states. First, we define the following *measures* on admissible configurations $S = \langle\langle E \mid \mathcal{K}\rangle, \gamma, \psi\rangle$:

- $m_1(S) = \{\!\{ \mid \psi(E') \mid \mid \ E' \in E \text{ and } \mid \psi(E') \mid > 0\}\!\}$,
- $m_2(S) = \Sigma_{E' \in Eqn_a(E)} \mid \psi(E') \mid$, where $Eqn_a(E)$ is the set of all equations and annotated equations of E,
- $m_3(S) =$ number of equations in E,
- $m_4(S) = |\gamma|$,
- $m_5(S) = \|E\|_\gamma$.

Intuitively, these orderings capture the following behavior of the unification algorithm: for any non-final admissible configuration $S = \langle P, \gamma, \psi\rangle$ with $P = \langle E \mid \mathcal{K}\rangle$ and any equation $E_1 \in E$, we can identify a new configuration $S' = \langle\langle E' \mid \mathcal{K}'\rangle, \gamma', \psi'\rangle$ and a transformation step $\varphi : P \Rightarrow_\sigma \langle E' \mid \mathcal{K}'\rangle$ which selects E_1, such that one of the following cases holds:

1. If we consider the subset of equations C (resp. C') of E (resp. E') made of all equations involving function symbols of Σ at the root, then $\{\!\{\psi(E'') \mid E'' \in C'\}\!\}$ is smaller than the multiset $\{\!\{\psi(E'') \mid E'' \in C\}\!\}$. In symbols, $m_1(S) > m_1(S')$. In this case, φ is either an (i)- or (d)-step.
2. The total number of equations involving function symbols of Σ at the roots of E is smaller than the total number of equations involving function symbols in Σ at the roots of E'. In symbols, $m_2(S) > m_2(S')$. In this case, φ is an (d)- or (i)-step, and it can be shown that $m_1(S) \geq m_1(S')$.
3. $m_3(S) > m_3(S')$. In this case, φ is an (an)-step, and it can be shown that $m_i(S) \geq m_i(S')$ for $i = 1, 2$.
4. $m_4(S) > m_4(S')$. In this case, φ is an (sg)-, (p)- or (i)-step, and it can be shown that $m_i(S) \geq m_i(S')$ for $i < 4$.
5. $m_5(S) > m_5(S')$. In this case, φ is an (fs)-, (fd)- or (d)-step, and it can be shown that $m_i(S) \geq m_i(S')$ for $i < 5$.

Now, it is quite easy to prove the *progress property* of the unification algorithm. We define the relation $\succ$ on $\mathtt{Cfg}$ as the lexicographic combination of the terminating orderings $>_1$, $>_2$, $>_3$, $>_4$, $>_5$, defined by $S >_i S'$ if and only if $m_i(S) > m_i(S')$. Note that $>_1$, $>_4$, and $>_5$ are defined via the multiset ordering on $2^{\mathbb{N}}$. Then, $\succ$ is also a terminating ordering.

Lemma A.2.3 (Progress property of a single unification step) *There exists a poset* $(\mathtt{Cfg}, \succ)$ *with* $\succ$ *a well-founded progress ordering, and a surjection* $\Im : \mathtt{Cfg} \to \mathtt{Adm}$*, such that, if* $P = \langle E \mid \mathcal{K} \rangle$ *is a non-final state,* W *is a finite set of variables, and* $\gamma \in [\![P]\!]$ *with* $Dom(\gamma) \subseteq \mathcal{V}$*, then there exist* $P' = \langle E' \mid \mathcal{K}' \rangle$*,* $\gamma' \in [\![P']\!]$*, and a transformation step* $P \Rightarrow_\sigma P'$*, with* $\Im(P) \succ \Im(P')$ *and* $\gamma = \sigma\gamma'$ $[W]$*.*

Proof We have $P \in \mathtt{Adm}$ because $S \in \mathtt{Cfg}$, and thus there exists an transformation step $\varphi' : P \Rightarrow_{\sigma'} P''$ with $P = \langle \{\!\{E_1, E\}\!\} \mid \mathcal{K} \rangle$ and E_1 the equation selected by φ' from E. By Lemma A.2.1, φ' is not a failure detection step.

1. If $E_1 \equiv s \approx t$ then $\psi(E_1) \equiv \frac{\mathcal{P}_1\,\mathcal{P}_2}{s\gamma == t\gamma}$, where $\mathcal{P}_1 \in \mathcal{PT}(s\gamma \approx u)$ and $\mathcal{P}_2 \in \mathcal{PT}(t\gamma \approx u)$ for some total λ-term u. We can choose the (an)-transformation step $\varphi : \langle \{\!\{s \approx t, E\}\!\} \mid \mathcal{K} \rangle \Rightarrow_\varepsilon P'$, where $P' = \langle s \approx_H t, E \mid \mathcal{K} \cup \{H\} \rangle$, and $S' = \langle P', \gamma', \psi \rangle$, where $\gamma' = \gamma \cup \{H \mapsto u\}$. Then, $S \in \mathtt{Cfg}$ and $\sigma\gamma' = \gamma' = \gamma$ $[W]$, because $H \notin W$. Moreover, $S \succ S'$ because $m_1(S) = m_1(S')$, $m_2(S) = m_2(S')$, and $m_3(S) > m_3(S')$.

2. If $E_1 \equiv s \approx_u t$ then $\psi(E_1) \equiv \frac{\mathcal{P}_1\,\mathcal{P}_2}{s\gamma \approx t\gamma}$, where $\mathcal{P}_1 \in \mathit{Soln}(s\gamma \approx u\gamma)$, $\mathcal{P}_2 \in \mathit{Soln}(t\gamma \approx u\gamma)$, and $u\gamma$ is a total λ-term. There are two possibilities:

   (a) At least one of the terms s, t is flex. We can assume without loss of generality that s is flex.

      i. If t is flex then $E_1 \equiv \lambda\overline{x_k}.X(\overline{y_p}) \approx_H \lambda\overline{x_k}.Y(\overline{y'_q})$ with $X, Y \in \mathcal{V}$ and φ' is (fs) or (fd) of the form $\langle \{\!\{E_1, E\}\!\} \mid \mathcal{K} \rangle \Rightarrow_\sigma \langle \{\!\{E\}\!\}\sigma \mid (\mathcal{K} \cup \{X, Y\})\sigma \rangle$. Suppose $X\sigma = \lambda\overline{y_p}.Z(\overline{z_q})$. Then, $\lambda\overline{x_k}.X(\overline{y_p})\gamma \approx H\gamma$ and $\lambda\overline{x_k}.Y(\overline{y'_q})\gamma \approx H\gamma$, because $\gamma \in \mathit{Soln}(E_1)$. We can conclude $\lambda\overline{x_k}.X(\overline{y_p})\gamma = \lambda\overline{x_k}.Y(\overline{y'_q})\gamma = H\gamma$ a total λ-term. Let $\gamma' = \gamma\restriction_{Dom(\gamma)\setminus\{X,Y\}} \cup \{Z \mapsto \lambda\overline{z_q}.H(\overline{x_k})\gamma\}$, $\varphi = \varphi'$, and $S' = \langle P', \gamma', \psi' \rangle$, where $\psi'(E'\sigma) = \psi(E')$ for all $E' \in \{\!\{E\}\!\}$. Then, $S' \in \mathtt{Cfg}$, $\gamma = \sigma\gamma'$ $[W]$, and $S \succ S'$ because $m_i(S) = m_i(S')$ for $i < 4$, $m_4(S) \geq m_4(S')$, and $m_5(S) > m_5(S')$.

      ii. If t is rigid then we can assume $s \equiv \lambda\overline{x_k}.X(\overline{s_m})$ and $t \equiv \lambda\overline{x_k}.a(\overline{t_n})$ with $a \in \Sigma \cup \{\overline{x_k}\}$. Since $\mathcal{P}_2 \in \mathcal{PT}(t\gamma \approx u\gamma)$ then $a = hd(t\gamma) = hd(u\gamma)$, and thus $u\gamma = \lambda\overline{x_k}.a(\overline{u_n})$. For $i = 1, \ldots, n$, we define the proofs $\tilde{\mathcal{P}}_i \in \mathcal{PT}(\lambda\overline{x_k}.t_i\gamma \approx \lambda\overline{x_k}.u_i)$ as follows: $\mathcal{P}_2 \equiv \frac{\tilde{\mathcal{P}}_1 \;\ldots\; \tilde{\mathcal{P}}_n}{t\gamma \approx u\gamma}$ and $\tilde{\mathcal{P}}_i = \lambda\overline{x_k}.t_i\gamma \approx \lambda\overline{x_k}.u_i$. If $u{\downarrow_\eta} = H \in \mathcal{FV}$ then we can perform the (sg)-step φ :

$P \Rightarrow_\sigma P'$, which selects E_1 and computes $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$. Let $S' = \langle P', \gamma', \psi' \rangle$, where $\gamma' = \gamma\restriction_{Dom(\gamma)\setminus\{H\}} \cup \{\overline{H_n \mapsto H\gamma\restriction_{1^k.n}}\}$, $\psi'(s \approx_{H\sigma} t) = \psi(E_1)$, and $\psi'(E') = \psi(E')$ if $E' \in \{\!\{E\}\!\}$. Then $S' \in \mathtt{Cfg}$, $\gamma = \sigma\gamma'$ $[W]$, and $S \succ S'$ because $m_i(S) = m_i(S')$ for $i < 4$ and $m_4(S) > m_4(S')$. Otherwise, $u = \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})$ such that $H_i\gamma = \lambda\overline{x_k}.u_i$ for $i = 1, \ldots, n$. If $\mathcal{P}_1 \in \mathcal{PT}(s\gamma \approx u\gamma)$ then $hd(s\gamma) = hd(u\gamma) = a$. In this case, we define, for $i = 1, \ldots, n$, the proofs $\mathcal{P}_i^* \in \mathcal{PT}(s\gamma\restriction_{1^k.i} \approx \lambda\overline{x_k}.u_i)$ as follows: $\mathcal{P}_1 \equiv \frac{\mathcal{P}_1^* \ldots \mathcal{P}_n^*}{s\gamma \twoheadrightarrow u\gamma}$ and $\mathcal{P}_i^* = s\gamma\restriction_{1^k.i} \approx \lambda\overline{x_k}.u_i$. If $X\gamma = \lambda\overline{y_m}.y_k(\overline{s'_q})$ then we can perform the (p)-transformation step $\varphi : P \Rightarrow_\sigma P'$, which computes $\sigma = \{X \mapsto \lambda\overline{y_m}.y_k(\overline{X_q(\overline{y_m})})\}$, and choose the admissible configuration $S' = \langle P', \gamma', \psi \rangle$, where $\gamma' = \gamma\restriction_{Dom(\gamma)\setminus\{X\}} \cup \{\overline{X_q \mapsto X\gamma\restriction_{1^m.q}}\}$. Then, $\gamma = \sigma\gamma'$ $[W]$ and $S \succ S'$, because $m_i(S) = m_i(S')$ for $i < 4$ and $m_4(S) > m_4(S')$. Otherwise, $X\gamma = \lambda\overline{y_m}.a(\overline{s'_n})$ with $a \notin \{\overline{x_k}\}$. In this case we can perform the (i)-transformation step $\varphi : P \Rightarrow_\sigma P'$ which selects E_1 and computes $\sigma = \{X \mapsto \lambda\overline{y_m}.a(\overline{X_n(\overline{y_m})})\}$ and $P' = \langle \{\!\{\overline{s\sigma\restriction_{1^k.n} \approx_{H_n} t\sigma\restriction_{1^k.n}}, E\sigma\}\!\} \mid (\mathcal{K} \cup \{X\})\sigma \rangle$. Let $S = \langle P', \gamma', \psi' \rangle$ where $\gamma' = \gamma \cup \{\overline{X_n \mapsto X\gamma\restriction_{1^m.n}}\}$, and $\psi'(s\sigma\restriction_{1^k.i} \approx_{H_i} t\sigma\restriction_{1^k.i}) = \frac{\mathcal{P}_i^* \tilde{\mathcal{P}}_i}{s\gamma\restriction_{1^k.i} \approx t\gamma\restriction_{1^k.i}}$ for $i = 1, \ldots, n$, $\psi'(E'\sigma) = \psi(E')$ if $E' \in E$. It is easy to check that $S' \in \mathtt{Cfg}$, $\gamma = \sigma\gamma'$ $[W]$, and $S \succ S'$, because $m_1(S) \geq m_1(S')$, $m_i(S) = m_i(S')$ for $i = 2, 3$, and $m_4(S) > m_4(S')$.

If $u\downarrow_\eta = H$ the φ' is an (sg)-transformation step. Then, $a = hd(t\gamma) = hd(u\gamma)$, and thus $u\gamma = \lambda\overline{x_k}.a(\overline{u_n})$. In this case, we consider the (sg)-transformation step $\varphi : P \Rightarrow_\sigma P'$, which selects E_1 and computes $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$. Let $S' = \langle P', \gamma', \psi' \rangle$, where $\gamma' = \gamma\restriction_{Dom(\gamma)\setminus\{H\}} \cup \{\overline{H_n \mapsto H\gamma\restriction_{1^k.n}}\}$, $\psi'(s \approx_{H\sigma} t) = \psi(E_1)$, and $\psi'(E') = \psi(E')$ if $E' \in \{\!\{E\}\!\}$. Then, $S' \in \mathtt{Cfg}$, $\gamma = \sigma\gamma'$ $[W]$, and $S \succ S'$, because $m_i(S) = m_i(S')$ for $i < 4$ and $m_4(S) > m_4(S')$.

(b) Both s and t are rigid. Then, $\mathcal{P}_1 \in \mathcal{PT}(s\gamma \approx u\gamma)$ and $\mathcal{P}_2 \in \mathcal{PT}(t\gamma \approx u\gamma)$, where $s = \lambda\overline{x_k}.a(\overline{s_n})$, $t = \lambda\overline{x_k}.a(\overline{t_n})$, $a \in \mathcal{F} \cup \{\overline{x_k}\}$, and $u\gamma = \lambda\overline{x_k}.a(\overline{u_n})$. If $u\downarrow_\eta = H$ then we perform the (sg)-transformation step $\varphi : P \Rightarrow_\sigma P'$, which selects E_1 and computes $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$. Let $S' = \langle P', \gamma', \psi \rangle$, where $\gamma' = \gamma\restriction_{Dom(\gamma)\setminus\{H\}} \cup \{\overline{H_n \mapsto H\gamma\restriction_{1^k.n}}\}$. Then $S' \in \mathtt{Cfg}$, $\gamma = \sigma\gamma'$ $[W]$, and $S \succ S'$, because $m_i(S) = m_i(S')$ for $i < 4$ and $m_4(S) > m_4(S')$. Otherwise, $u \equiv \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})$. For $i = 1, \ldots, n$, we define the proofs $\mathcal{P}_i^* \in \mathcal{PT}(\lambda\overline{x_k}.s_i\gamma \approx \lambda\overline{x_k}.u_i)$ and $\tilde{\mathcal{P}}_i \in \mathcal{PT}(\lambda\overline{x_k}.t_i\gamma \approx \lambda\overline{x_k}.u_i)$ as follows:

- $\mathcal{P}_1 \equiv \frac{\mathcal{P}_1^* \ \ldots \ \mathcal{P}_n^*}{s\gamma \approx u\gamma}$; If $s\gamma = u\gamma$ we define $\mathcal{P}_i^* = \lambda\overline{x_k}.s_i\gamma \approx \lambda\overline{x_k}.u_i$.
- $\mathcal{P}_2 \equiv \frac{\tilde{\mathcal{P}}_1 \ \ldots \ \tilde{\mathcal{P}}_n}{s\gamma \approx u\gamma}$; If $t\gamma = u\gamma$ we define $\tilde{\mathcal{P}}_i = \lambda\overline{x_k}.t_i\gamma \approx \lambda\overline{x_k}.u_i$.

In this case, we perform the (d)-transformation step $\varphi : P \Rightarrow_\sigma P'$, which selects E_1 and computes $\sigma = \varepsilon$. The corresponding admissible configuration is $S' = \langle P', \gamma, \psi' \rangle$, where $\psi'(\lambda\overline{x_k}.s_i \approx_{H_i} \lambda\overline{x_k}.t_i) = \frac{\mathcal{P}_i^* \tilde{\mathcal{P}}_i}{\lambda\overline{x_k}.s_i\gamma == \lambda\overline{x_k}.t_i\gamma}$ for $i = 1, \ldots, n$, $\psi'(E') = \psi(E')$ if $E' \in E$. We have $S \succ S'$ because $m_i(S) \geq m_i(S')$ for $i < 5$ and $m_5(S) > m_5(S')$.

□

We are now ready to prove the main result presented in this Appendix: *soundness* and *completeness* of the higher-order unification algorithm.

Theorem A.2.4 (Properties of the Higher-Order Unification Algorithm)

(1) **Soundness***: Let $\langle E \mid \emptyset \rangle \Rightarrow_\sigma^* P$ be an unification-derivation. Then, $\sigma\gamma \in Soln(E)$ whenever $\gamma \in [\![P]\!]$.*

(2) **Completeness***: Let E be a set of strict equations. Then, $\mathcal{A}(E) = \{\gamma\restriction_{\mathcal{FV}(E)} \mid \gamma \in Soln(E)\}$.*

Proof

- Let $\gamma \in [\![P]\!]$. We prove by induction on the length of $\Pi : \langle E \mid \emptyset \rangle \Rightarrow_\sigma^* P$ that $\sigma\gamma \in Soln(E)$. If $|\Pi| = 0$ then $\sigma = \varepsilon$ and $\sigma\gamma = \gamma \in [\![P]\!] = [\![\langle E \mid \emptyset \rangle]\!] = Soln(E)$. If $|\Pi| > 0$ then we can write $\Pi : \langle E \mid \emptyset \rangle \Rightarrow_{\sigma_1}^* P_1 \Rightarrow_{\sigma_2} P$. By Lemma A.2.1, we know that $\sigma_2\gamma \in [\![P_1]\!]$. We can now apply the induction hypothesis to the shorter derivation $\langle E \mid \emptyset \rangle \Rightarrow_{\sigma_1}^* P$ and learn that $\sigma\gamma = \sigma_1\sigma_2\gamma \in [\![\langle E \mid \emptyset \rangle]\!] = Soln(E)$.

- Since $\mathcal{A}(E) \subseteq Soln(E)$ by *soundness*, we must only show that $Soln(E) \subseteq \mathcal{A}(E)$. Let $\gamma \in Soln(E)$, $P = \langle E \mid \emptyset \rangle$, and $W_0 = \mathcal{FV}(E) \cup Dom(\gamma)$. First, we prove that, for every given admissible state P, any finite set of variables W, and $\gamma \in [\![P]\!]$, there exists an unification-derivation $\Phi : P \Rightarrow_\sigma^* P'$ such that $\gamma = \sigma\gamma'\ [W]$ for some $\gamma' \in [\![P']\!]$. The proof is by induction with respect to the well-founded progress ordering $\succ$ introduced in Lemma A.2.3. If P is final then we can choose $\Phi : P \Rightarrow_\varepsilon^0 P$ and $\gamma' = \gamma$. Otherwise, we can apply Lemma A.2.3 to determine $P_1 = \langle E_1 \mid \mathcal{K}_1 \rangle$, $\gamma_1 \in [\![P_1]\!]$, and a transformation step $\varphi : P \Rightarrow_{\sigma_1} P_1$ with $\Im(P) \succ \Im(P_1)$ and $\gamma = \sigma_1\gamma_1\ [W]$. Let $W' = W \cup \mathcal{FV}(\{X\sigma_1 \mid X \in W\})$. By induction hypothesis for $\Im(P_1)$, there exists an unification-derivation $\Phi' : P_1 \Rightarrow_{\sigma'}^* P'$ such that $\gamma_1 = \sigma'\gamma'\restriction_{W'}$ for some $\gamma' \in [\![P']\!]$. Let $\sigma = \sigma_1\sigma'$ and Φ the unification-derivation obtained by prepending φ to Φ'. Then, $\Phi : P \Rightarrow_\sigma^* P'$ and $\sigma\gamma'$ is a computed answer. Also, $\gamma\restriction_W = \sigma_1\gamma_1\restriction_W = \sigma_1\sigma'\gamma'\restriction_W = \sigma\gamma'\restriction_W$, and this concludes our preliminary proof. In particular, if $\gamma \in Soln(E)$ then $\gamma \in [\![P]\!]$ where $P = \langle E \mid \emptyset \rangle$. According to our preliminary result, there exists an unification-derivation $\Phi : P \Rightarrow_\sigma^* P'$ such that $\gamma = \sigma\gamma'\ [\mathcal{FV}(E)]$ for some $\gamma' \in [\![P']\!]$. Thus, $\sigma\gamma'\restriction_{\mathcal{FV}(E)} \in \mathcal{A}(E)$, $\sigma\gamma'\restriction_{\mathcal{FV}(E)} = \gamma\restriction_{\mathcal{FV}(E)}$, and $\gamma \in \mathcal{A}(E)$.

□

Bibliography

[1] S. Abramsky and A. Jung. *Handbook of Logic in Computer Science*, volume 3, chapter Domain Theory, pages 1–168. Clarendon Press, Oxford, 1994.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, pages 268–279, 1994.

[3] S. Antoy and M. Hanus. Functional logic programming. In *Communications of the ACM*, 2010.

[4] K. R. Apt. *Handbook of theoretical computer science*, chapter 10, pages 495–574. Elsevier Science Publishers, 1990.

[5] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[6] H. P. Barengredt. *The Lambda Calculus. Its syntax and semantics.* Elsevier, revised edition, 1984.

[7] B. Beckert. *Automated deduction - A basis for applications*, chapter Rigid E-Unification, pages 265–289. Springer, 1998.

[8] M. Bezem, J. Willem-Klop, and R. de Vrijer. *Term Rewriting Systems.* Cambridge University Press, 2003.

[9] G. Birkhoff. On the structure of abstract algebras. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 31, pages 433–454, 1935.

[10] R. V. Book and k. F. Otto. *String rewriting systems.* Springer, 1993.

[11] J. Boye, J. Maluszynski, and U. Nilsson. *Rewriting systems (Lecture notes)*, September 1999.

[12] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS'94)*, 16:1361–1398, 1994.

[13] A. Brogi and F. Turini. Fully abstract composition semantics for an algebra of logic programs. *Theoretical Computer Science*, 149:201–229, 1995.

[14] R. Caballero, F. J. López Fraguas, and M. Rodríguez Artalejo. A logical framework for the algorithmic debugging of lazy functional-logic programs. In *Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP'00)*, 2000.

[15] R. Caballero and M. Rodríguez Artalejo. A declarative debugging system for lazy functional logic programs. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, volume 64 of *Electronic Notes in Theoretical Computer Science*, 2001.

[16] R. Caballero and M. Rodríguez Artalejo. DDT: a declarative debugging tool for functional-logic languages. In *Proceedings of the Functional and Logic Programming, 7th International Symposium (FLOPS'04)*, 2004.

[17] A. Casas, D. Cabeza, and M. V. hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, 2006.

[18] T. R. Chuang and J. L. Lin. On modular transformation of structural content. In *Proceedings of the 2004 ACM symposium on Document engineering*. ACM, 2004.

[19] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1940.

[20] J. M. Cleva, J. Leach, and F. J. López Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 9–19, 2004.

[21] N. Cutland. *Computability. An introduction to recursion function theory*. Cambridge University Press, 1980.

[22] R. del Vado Vírseda. Estrategias de estrechamiento perezoso. Master's thesis, Universidad Complutense de Madrid, 1999.

[23] R. del Vado Vírseda. A higher-order demand-driven narrowing calculus with definitional trees. In *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, September 26-28, 2007, Proceedings*, pages 169–184, 2007.

[24] R. del Vado Vírseda. A higher order logical framework for the algorithmic debugging and verification of declarative programs. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9*, pages 49–60, 2009.

[25] R. del Vado Vírseda. Cooperation of algebraic constraint domains. In M. Johnoson and D. Pavlovic, editors, *Proceedings of the Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST 2010)*, volume 6486 of *LNCS*, pages 180–200, 2010.

[26] R. del Vado Vírseda and I. Castiñeiras Pérez. A theoretical framework for the declarative debugging of functional logic programs with lambda abstractions. In *Proceedings of the 18th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP2009)*, pages 162–178, 2009.

[27] R. del Vado Vírseda and F. Pérez Morente. A modular semantics for higher-order declarative programming with constraints. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'11), July 20-22, Odense, Denmark*, July 2011. To appear.

[28] G. Dowek. *Handbook of automated reasoning, volume 2*, chapter 16, pages 1009–1062. Elsevier Science Publishers, 2001.

[29] S. Estévez, T. Hortalá, M. Rodríguez, R. del Vado Vírseda, F. Sáenz, and A. Fernández. On the cooperation of constraint domains H, R and FD in CFLP. *Theory and Practice of Logic Programming (TPLP)*, 9(4):415–527, 2009.

[30] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69, 1989.

[31] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 102(1):86–113, 1993.

[32] M. Fernandez. *Models of computation - An introduction to computability theory.* Springer, UTics series, 2009.

[33] M. Fitting. Enumeration operators and modular logic programming. *Journal of Logic Programming*, 4(1):11–21, 1987.

[34] M. Fitting. Fixpoint semanticss for logic programming. a survey. *Theoretical Computer Science*, 278:25–51, 2002.

[35] M. Franssen. Implementing rigid e-unification. In *23nd International Workshop on Unification proceedings*, 2009.

[36] R. Giacobazzi. Abductive analysis of modular logic programs. *Journal of Logic Programming*, 8(4):457–483, 1998.

[37] W. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[38] J. C. González Moreno, M. T. Hortalá González, F. J. López Fraguas, and M. Rodríguez Artalejo. An approach to declarative programming based on a rewriting logic. *The journal of logic programming*, 40:47–87, 1999.

[39] J. C. González Moreno, M. T. Hortalá González, and M. Rodríguez Artalejo. A higher order rewriting logic for functional logic programming. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, 1997.

[40] J. C. González Moreno, M. T. Hortalá González, and M. Rodríguez Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 1:1–71, 2001.

[41] R. Haemmerlé and F. Fages. Modules for prolog revisited. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP'06)*, volume 3132 of *Lecture Notes in Computer Science*. Springer, 2006.

[42] M. Hamada. Strong completeness of a narrowing calculus for conditional rewrite systems with extra variables. In *In proceedings of Computing: the Australasian Theory Symposium (CATS 2000)*, 2000.

[43] C. Hankin. *An introduction to Lambda Calculi for Computer Scientists*. King's College Publications, 2004.

[44] M. Hanus. Curry: A truly integrated functional logic language (http://www-ps.informatik.uni-kiel.de/currywiki/).

[45] M. Hanus. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'97)*, 1997.

[46] M. Hanus. Multi-paradigm declarative languages. In *Proceeding of the 23rd International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 45–75, 2007.

[47] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1), 1999.

[48] R. Harper and F. Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic Computation*, 8:5–31, 1998.

[49] J. Herbrand. Recherches de la theorie de la demonstration. Master's thesis, Université de Paris, 1930.

[50] P. Hill and J. Lloyd. *The Gödel programming language.* MIT Press, 1994.

[51] J. R. Hindley. *Basic simple type theory.* Cambridge University Press, 1997.

[52] J. R. Hindley and J. P. Seldin. *Lambda-Calculus and Combinators, an introduction.* Cambridge University Press, 2008.

[53] P. Hudak. Conception, evolution and applicattion of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.

[54] J. W. Klop. *Term rewriting systems - Handbook of logic in computer science*, volume 2, chapter 1, pages 1–116. Oxford University Press, 1992.

[55] D. Kranzlmüller, C. Schaubschläger, M. Scarpa, and J. Volkert. A modular debugging infraestructure for parallel programs. In *Proceedings of the International Conference on Parallel Computing: Software Technology, Algorithms, Architectures & Applications ParCo 2003*, volume 13, pages 143–150, 2004.

[56] J. Lipton and S. Nieva. Higher-order logic programming languages with constraints: A semantics. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA 2007)*, 2007.

[57] J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag, 2nd (extended) edition, 1987.

[58] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, volume LNCS, pages 283–298, 2007.

[59] F. López Fraguas, E. Martin Martin, and J. Rodriguez Hortala. Lyberal typing for functional logic programs. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10)*, volume 6461 of *Lecture Notes in Computer Science*, pages 80–96, 2010.

[60] F. J. López Fraguas and J. Rodríguez Hortalá. The full abstraction problem for higher order functional-logic programs. In *Proceedings of The 19th Workshop on Logic-based methods in Programming Environments (WLPE'09)*, 2009.

[61] F. J. López Fraguas, J. Rodríguez Hortalá, and j. Sánchez Hernández. Rewriting and call-time choice: the ho case. In *Proceedings of the 17th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP2008)*, 2008.

[62] F. J. López Fraguas and j. Sánchez Hernández. Toy: A multiparadigm declarative system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, 1999.

[63] C. L. Lucchesi. The undecidability of the unification problem for third order languages. Technical report, University of Waterloo, Canada, 1972.

[64] P. Mancarella and D. Pedreschi. An algebra of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming (SLP'88)*, 1988.

[65] M. Marin. *Functional logic programming with distributed constraint solving.* PhD thesis, Universität Linz, 2000.

[66] A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on programming languages and systems (TOPLAS)*, 2:258–282, 1982.

[67] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192:3–29, 1998.

[68] J. Meseguer. Conditional rewriting logic as an unified model of concurrency. *Theoretical computer science*, 96:73–155, 1992.

[69] D. Miller. A theory of modules for logic programming. In *Proceedings of the 1986 International Symposium on Logic Programming (SLP'86)*, 1986.

[70] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, New York, NY, USA, 1991.

[71] D. Miller and G. Nadathur. Higher-order logic programming. In *Third International Conference on Logic Programming (ICLP'86)*, 1986.

[72] B. Möller. On the algebraic specification of infinite objects - ordered and continuous models of algebraic types. *Acta Informatica*, 22:537–578, 1985.

[73] J. M. Molina Bravo and E. Pimentel. Modularity in functional-logic programming. In *Proceedings of the Fourteenth International Conference on Logic Programming*, 1997.

[74] J. M. Molina Bravo and E. Pimentel. Composing programs in a rewritting logic for declarative programming. *Theory and Practice of Logic Programming (TPLP)*, 3:189–221, 2003.

[75] E. Monfroy and C. Castro. Basic components for constraint solver cooperations. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC'03)*, 2003.

[76] E. Monfroy and C. Castro. A component language for hybrid solver cooperations. In *Proceedings of the Third International Conference on Advances in Information Systems (ADVIS'04)*, 2004.

[77] L. Naish. Higher-order logic programming in prolog, 1995.

[78] L. C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[79] B. C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[80] P. Pietrzak, J. Correas, M. Hermenegildo, and G. Puebla. A practical type analysis for verification of modular prolog programs. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM'08)*, 2008.

[81] C. Prehofer. *Solving higher-order equations. From logic to programming*. Birkhäuser, 1999.

[82] X. Qi. *An Implementation of the Language Lambda Prolog Organized around Higher-Order Pattern Unification*. PhD thesis, UNIVERSITY OF MINNESOTA, 2009.

[83] Z. Qian. Higher-order equational logic programming. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'94)*, 1994.

[84] C. Reade. *Elements of functional programming*. Addison-Wesley, 1989.

[85] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic in Computer Science (SDP'85)*, 1985.

[86] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[87] M. Rodríguez Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics*. Springer-Verlag, 2001.

[88] D. Sanella and L. Wallen. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12(1-2):147–177, 1992.

[89] D. S. Scott. Continuous lattices. In F. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, number 274 in Lecture Notes in Mathematics, pages 97–136, Berlin, 1972.

[90] D. S. Scott. Domains for denotational semantics. In *Proceedings of the International Conference on Automata, Languages and Programming (ICALP'82)*, pages 577–613, 1982.

[91] G. Smolka. The Oz programming language and system. In *Concurrency and Parallelism, Programming, Networking, and Security*. Springer Berlin / Heidelberg, 1996.

[92] Z. Somogy, F. Henderson, and T. Conwat. The implementation of mercury: an efficient purely declarative logic programming language. In *Proceedings of the Workshop on Implementation Techniques for Logic Programming Languages (ILPS '94)*, 1994.

[93] D. H. Warren. Higher-order extensions to prolog: are they needed? *Machine Intelligence*, 10:441–454, 1982.